ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación :

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

Título del proyecto:

Motor de Juego Multijugador en red

Miguel Garde Vallés

Mikel Izal Azcárate

Pamplona, 2 de Julio de 2010

ÍNDICE

A Mikel Izal, por haberme guiado y aguantado durante todo un año.

A Adrián Equísoain, por crear las animaciones más infantiles y gores posibles.

A Roberto Carlos Laita, por componer las repetitivas y pegadizas melodías que tan profundamente me han perforado la mente.

Al equipo Team17, por "prestarme" los efectos de sonido del fabuloso Worms Armageddon.

A los ludópatas de la sala de ordenadores de la residencia Fuerte del Príncipe, por prestarse a hacer de Beta Testers y conejillos de indias en general.

1 – RESUMEN

Este proyecto consiste en la creación de un motor de juego multijugador en red utilizando el lenguaje de programación Java. La finalidad del mismo es, aparte de hacer un juego, conseguir separar el motor de juego y el de red para fines didácticos. El desarrollo se ha dividido en diferentes etapas para un mejor resultado: en primer lugar se ha hecho un sencillo análisis de requisitos y diseño general, estableciendo los objetivos a alcanzar, qué clase de videojuego se va a crear y cuál va a ser su mecánica, cuál va a ser su temática (la historia que va a contar), qué partes va a tener y cómo se deberán hacer, los elementos dentro del juego, la estructura de los escenarios, el núcleo interno, etc. Tras esto, se ha pasado a implementarlo, estableciendo con detenimiento qué clases hay, sus relaciones, tanto de herencia como de comunicación, y su funcionalidad. También en esta parte se ha desarrollado un protocolo de red que permitiera la comunicación entre el cliente y el servidor, tanto a la hora de buscar y crear partidas como durante el juego. Después de la implementación, se ha realizado un estudio de rendimiento mediante una serie de pruebas, con gráficas comparativas de diferentes resultados según factores como la calidad de la conexión, la potencia del ordenador, etc. teniendo en cuenta los diferentes modos de juego (un jugador o multijugador). Para terminar, el proyecto ha finalizado con una serie conclusiones en las que se ha valorado el trabajo realizado y el resultado obtenido, así como posibles ampliaciones y novedades que se pudieran incluir en un futuro, tanto en el aspecto de juego (añadir más escenarios y enemigos, vídeos, etc.) como en el aspecto técnico (hacer un estudio más intensivo del rendimiento, corregir algunos errores que hayan quedado, etc.).

2 – INTRODUCCIÓN

El software de entretenimiento o videojuegos constituye hoy en día, sin lugar a dudas, la industria más importante dentro del mundo del entretenimiento, superando desde hace ya varios años a la industria cinematográfica. Se caracteriza por el uso de tecnologías punteras y por liderar la investigación en campos tan importantes como la computación gráfica, interfaces, usabilidad, simulaciones e Inteligencia Artificial entre otros.

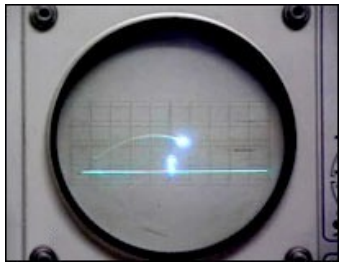
Se han convertido, así mismo, en una forma de expresión cultural muy fuerte, con personajes y figuras mundialmente reconocidas y estilos definibles de acuerdo al origen de sus diseñadores.

Aunque aparentemente los videojuegos parezcan "tonterías para jugar y nada más", suponen muchas veces un reto y son, con frecuencia, más complejos que aplicaciones cotidianas como un navegador, un IDE o un escritorio.

2.1 - Inicios

Durante bastante tiempo ha sido complicado señalar cual fue el primer videojuego, principalmente debido a las múltiples definiciones de éste que se han ido estableciendo, pero se puede considerar como primer videojuego tanto el *Nought and crosses*, también llamado OXO, desarrollado por Alexander S. Douglas en 1952, como el Tennis for Two, creado en 1957 por William Higginbotham. El primero era una versión computarizada del “tres en raya” que permitía enfrentar a un jugador humano contra la máquina; el segundo fue el primero en permitir el juego entre dos jugadores humanos. La controversia estriba en que el OXO no tenía ningún tipo de movimiento ni gráficos dentro del sistema, mientras que Tennis for Two sí. Como contrapartida, éste no utilizaba monitores convencionales de CRT o televisión, mientras que aquél sí los usaba.

Captura del OXO.



Tennis for Tow en un osciloscopio.

Posteriormente, en 1972 se fundó Atari, la que puede ser considerada la primera empresa del sector; tres años después, la empresa sacó al mercado lo que sería considerado la piedra angular de la industria del videojuego: la máquina recreativa Pong. Al año siguiente, en 1976, aparecieron las primeras aventuras de texto, basadas en la toma de decisiones y solución de acertijos en vez de en reflejos, como se basaban los demás.

Aventura gráfica: Monkey Island.

En la década de los 80 hubo un fuerte crecimiento en el sector del videojuego debido a la popularidad de las máquinas recreativas y videoconsolas, aparecidas en la década anterior. La empresa Nintendo lanzó su primera consola, NES, en 1983; y dos años después apareció Super Mario Bros. Éste supuso un punto de inflexión en el desarrollo de videojuegos, pues a diferencia de los anteriores, que consistían en varias pantallas repetitivas en las que se intentaba obtener la mejor puntuación, el nuevo juego desarrollado tenía un objetivo y un final.

Otra rama que también creció con fuerza fue la de los videojuegos portátiles, alcanzando la cúspide con el lanzamiento de GameBoy, en 1989.

Máquina SNES europea.

Desde los 90 en adelante, la mejora progresiva del hardware permitió al sector de los videojuegos dar un importante salto cualitativo tanto en el aspecto visual, apareciendo los juegos de 16, 32 y 64 bits y posteriormente, gracias a las aceleradoras 3D, los juegos tridimensionales; como en la jugabilidad, permitiendo el modo multijugador gracias a internet.

Juego MMORPG World of Warcraft.

2.2 - Géneros de Videojuegos

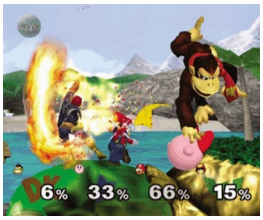
Los videojuegos se pueden clasificar como un género u otro dependiendo de su representación gráfica, el tipo de interacción entre el jugador y la máquina, la ambientación y su sistema de juego, siendo este último el criterio más habitual a tener en cuenta. Hay que decir que cada vez es más es habitual que un videojuego contenga elementos de diversos géneros, cosa que hace difícil su clasificación. A continuación se presenta una de las posibles clasificaciones:

Acción

LUCHA

Recrean combates entre personajes controlados tanto por el jugador como por la máquina, viéndose éstos desde una perspectiva lateral. En este tipo de videojuegos es común el uso de artes marciales, tanto reales como ficticias; armas blancas, espadas, martillos, etc. y ataques a distancia, generalmente de carácter mágico.

Ejemplo: Super Smash Bros (Nintendo), Street Fighter (Capcom), Soul Calibur (Namco).



BEAT 'EM UP

Son videojuegos similares a los anteriores, con la diferencia de que aquí, los jugadores deben combatir contra un gran número de individuos a lo largo de diferentes niveles. Normalmente pueden jugar dos o más personas simultáneamente de forma cooperativa.

Ejemplo: Final Fight (Capcom), King of Monsters (SNK), Captain Commando (Capcom).

DISPAROS

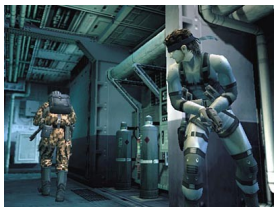
Consisten en mover al personaje y utilizar armas de fuego para abatir a los enemigos. Hay

tres subgéneros principales: en primera persona, o FPS, se enfatiza en la precisión (se tiene el punto de vista del personaje); son juego tales como Doom o Counter-Strike. El segundo subgénero es en tercera persona (la cámara se sitúa justo detrás del personaje), en éste se sacrifica la precisión para permitir una mayor libertad de movimiento; ejemplos de este tipo son el Grand Theft Auto o el Gears of War. Finalmente existe un subgénero llamado shoot 'em up, bastante diferente a los otros dos; aquí hay una perspectiva lateral (suelen ser juegos en 2D) y los niveles están llenos de proyectiles tanto propios como del enemigo; juegos como r-Type o Metal Slug son un claro ejemplo.

SIGILO

Aunque pueden incluirse como un subgénero de los juegos de disparo, los de sigilo se basan en la furtividad y la estrategia en vez de buscar la confrontación directa con el enemigo.

Ejemplo: Metal Gear (Konami), Splinter Cell (Ubisoft), Thief (Eidos Interactive).



ESTRATEGIA

Se caracterizan por obligar al jugador a pensar bien lo qué hacer, así como por implementar una buena inteligencia artificial. Hay dos subgéneros: por turnos y en tiempo real o RTS. En el primero el jugador tiene un rival, igual que él, y se van turnando para la realización de movimientos con el fin de ganar al contrario; ejemplos de este tipo de juego son el Ajedrez o el Ogre Battle. Los segundos enfatizan la planificación y el manejo de materias primas, operaciones militares y tecnologías; por ejemplo el Empire Earth, Civilization o Warcraft.

PLATAFORMAS

En este tipo de videojuegos el jugador controla a un personaje que debe avanzar por el escenario evitando obstáculos físicos, ya sea saltando, escalando o agachándose. Inicialmente los personajes se movían por niveles con un desarrollo horizontal, pero con la llegada de los gráficos 3D este desarrollo se ha ampliado hacia todas las direcciones posibles.

Ejemplos: Super Mario (Nintendo), Sonic the Hedgehog (Sonic Team), Rayman (Ubisoft).

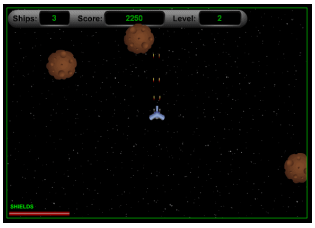
SIMULACIÓN

Este género se caracteriza por marcar un aspecto de la vida real, llevada a un juego de la forma más fiel posible (ya sea en cuanto a capacidades de decisión, representación gráfica o comportamiento físico), donde se tiene total control de lo que pasa. En muchos de ellos se implica tanto al jugador, que se consigue hacerle creer que lo que está pasando es real. Hay varios subgéneros: en primer lugar están los simuladores de combate, en los que el jugador utiliza armamento de forma realista; un ejemplo es el Armed Assault. Otro tipo de simuladores son los de conducción, ya sea de coches, aviones, etc, como Flight Simulator. También hay simuladores de construcción, ya sea de ciudades, parques o zoos; por ejemplo, Sim City. Finalmente, existen los simuladores de vida donde se controlan todos los aspectos de un personaje: sus emociones, su alimentación, sus relaciones, etc. The Sims o Spore son juegos de este tipo.

ARCADE

Se caracterizan por la simplicidad, son de acción rápida y de jugabilidad. No requiere historia, son juegos largos o repetitivos en los que cuenta la puntuación conseguida o el tiempo que haya costado pasárselo.

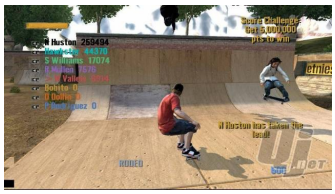
Ejemplo: Pac-Man (Namco), Space Invaders (Taito), Asteroids (Atari).



DEPORTE

Simulan juegos de deporte real, entre ellos encontramos golf, tenis, fútbol, hockey, juegos olímpicos, etc. El propósito es el mismo que el deporte original.

Ejemplo: FIFA 09 (EA Sports), Tony Hawk's (Neversoft), NBA Live 09 (EA Sports).



CARRERAS

Consisten en comenzar en un punto y llegar a una meta antes que los contrincantes, normalmente en vehículos, añadiendo en algunos casos trampas para ralentizarlos.

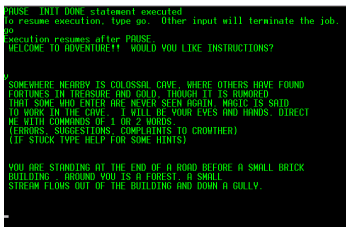
Ejemplo: F-Zero (Nintendo), Need for Speed (Electronic Arts), Mario Kart (Nintendo).

Aventura

AVENTURA CONVERSACIONAL

Fueron los primeros que se vendieron en el mercado. En ellos el jugador encarna a un protagonista que por lo general debe resolver incógnitas y rompecabezas con objetos diversos. Se usa el teclado para introducir órdenes y el ordenador describe lo que pasa.

Ejemplo: Zork (Infocom), Colossal Cave Adventure (Will Crowther), Enchanter (Infocom).



AVENTURA GRÁFICA

Es el sucesor de la aventura conversacional; el sistema de juego es el mismo, se deben resolver misterios con objetos, pero a diferencia del anterior, éste utiliza representación gráfica en vez de textual y se sirve del ratón para interaccionar (point and click) en lugar de introducir comandos.

Ejemplos: Day of the Tentacle (LucasArts), Broken Sword (Revolution Software), Monkey Island (LucasArts).

ROL

Se caracterizan por la interacción con el personaje, una historia profunda y una evolución del mismo a medida que la historia avanza. Para lograr la evolución generalmente se hace que el jugador se enfrasque en una aventura donde irá conociendo nuevos personajes, explorando el mundo para ir juntando armas, experiencia, aliados e incluso magia. Los RPG clásicos, inspirados en los juegos de tablero, realizan las batallas por turnos, es decir, el jugador usa su equipo y habilidades aprendidas para atacar mediante una serie de comandos y después debe quedar estático y esperar a recibir el ataque del otro jugador o CPU. Actualmente se han puesto de moda los RPG en línea o MMORPG (*Massive Multiplayer Online Rol Playing Game*) donde cada jugador crea un personaje y mediante una conexión a internet, entra a un mundo donde miles de jugadores se unen a la aventura, exploran, intercambian y evolucionan juntos.

Ejemplo: Tibia (ClipSoft), Final Fantasy (Square Enix), World of Warcraft (Blizzard).

Otros

AGILIDAD MENTAL

El objetivo aquí es resolver ejercicios con dificultad progresiva para desarrollar la habilidad mental.

Ejemplo: Brain Training (Nintendo), Tetris (Alexey Pajitnov), WarioWare, Inc. (Nintendo).

EDUCATIVOS

Enseñan mientras promueven diversión o entretenimiento (Didactismo). A diferencia de una enciclopedia, tratan de entretener mientras se memorizan conceptos o información.

Ejemplo: Fine Artist (Microsoft), Ven a Jugar con Pipo (Cibal Multimedia), Katomic (KDE).

ESCRITORIO

Este tipo de juegos son los utilizados por la mayoría de la gente cuando está ociosa frente a un ordenador. Los más conocidos son los juegos de cartas, aunque existen algunas otras variantes. Se caracterizan por su simplicidad en cuanto a gráficos, sonidos y jugabilidad.

Ejemplo: Buscaminas, Solitario, Mahjong.

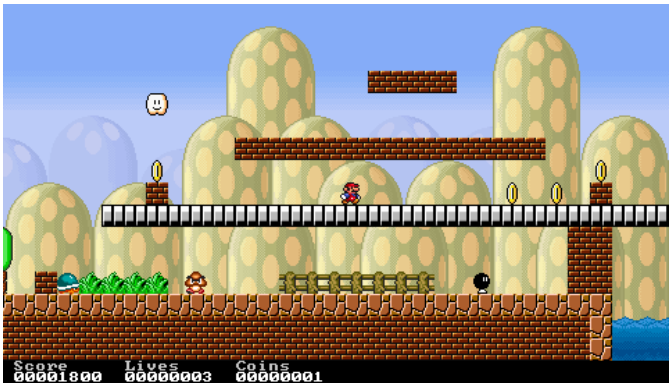
3 – DISEÑO

En este apartado se definirá la idea del videojuego, qué se quiere conseguir, para qué se quiere conseguir y cómo se va a conseguir. Así mismo, se especificarán todos los contenidos del mismo haciendo un análisis de lo que se quiere hacer y posteriormente se hará un diseño general que permita una futura implementación.

3.1 - Descripción inicial del Juego

El tipo de aplicación que se pretende hacer es un juego en dos dimensiones (2D) con movimiento lateral que combina la habilidad necesaria en un juego de plataformas típico (como uno del tipo de Mario Bros) con la destreza y reflejos de un Shoot'em up, como el Metal Slug; con un añadido adicional, el propio jugador puede modificar de forma dinámica el escenario, pudiendo alcanzar zonas del mapa en un principio inaccesibles, cambiando la trayectoria de movimiento de los enemigos, etc. De este modo, los mapas se convierten en puzzles que el jugador debe resolver mientras está pendiente de las amenazas enemigas que existen en el entorno.

Como complemento añadido, se podrá jugar de forma cooperativa o competitiva en red de área local, siendo la aplicación capaz de comunicarse a través de sockets tanto para anunciar la creación de las partidas como para enviar y recibir eventos durante el juego.



Escena de un juego de plataformas con un mapa estructurado en tiles.

Escena de un Shoot'em up con jugadores y enemigos que disparan.

3.1.1 - Temática del Juego

La historia dentro del videojuego consiste en unas instalaciones secretas en las que están haciendo experimentos con armamento y bebés. El fin de los mismos es obtener soldados físicamente perfectos capaces de realizar intrusiones eficaces aprovechando las capacidades de esas armas.

JUEGO DE UN JUGADOR

El jugador encarna a uno de los bebés, que consigue hacerse con una de las armas e intenta escapar del lugar. Para conseguirlo debe aprovechar todos los recursos disponibles al máximo, pues la munición del arma se acaba enseguida y el peligro acecha en todo momento; el bebé deberá enfrentarse a todo tipo de criaturas experimentales. El objetivo en cada nivel es llegar con vida hasta un punto concreto del escenario, llamado meta; una vez alcanzada, se pasa al siguiente nivel.

JUEGO EN RED COOPERATIVO

La temática y el objetivo son los mismos que antes, solo que no escapa un único bebé, sino varios. Cada uno de ellos controlado por un jugador.

JUEGO EN RED COMPETITIVO

Existe una sala de pruebas dentro de un laboratorio en la que obligan a los bebés a matarse entre sí. Los jugadores deben luchar entre ellos por ser los únicos supervivientes. Cuando queda un único personaje en pie, se le considera el ganador y la partida vuelve a empezar.

3.1.2 - El Juego: Elementos y Escenarios

Los escenarios, inspirados en los juegos de plataformas, están formados por tiles o cuadrados adyacentes que componen el suelo y paredes sobre los que caminan las criaturas. Los mapas tienen también unas imágenes de fondo que utilizan la técnica de parallax scrooling, o fondo de paralaje, para dar sensación de profundidad mediante un lento movimiento de la imagen comparado con el avance del propio escenario. La carga de los mapas se hará leyendo los datos de un fichero de texto, sirviendo cualquier editor de texto como editor de los niveles.

La ambientación del juego será ligeramente infantil, visualmente muy colorida y con música alegre y rítmica, para contrastar con las muertes de las criaturas, mostradas de forma violenta y sangrienta.

PROTAGONISTA

El jugador principal estará representado por un bebé con una pistola en la mano. Está inspirado en los Shoot'em up: podrá saltar, avanzar lateralmente en ambos sentidos, disparar diferentes tipos de munición e incluso podrá suicidarse (idea cogida de otro juego de plataformas: *Splosion Man*). Morirá al contactar con cualquier enemigo o con un proyectil.

PROYECTILES: BLOQUES

Cada uno de los proyectiles disparados por el jugador será un elemento (llamado bloque) que avanzará en línea recta perdiendo velocidad poco a poco y con la capacidad de destruir casi cualquier criatura que esté en la trayectoria (incluso al propio jugador). Tras detenerse pasará a formar parte del escenario, pudiendo utilizarse para acceder a distintos lugares del mapa. El usuario podrá también inmaterializar todos los bloques a voluntad, pasando a ser semitransparentes y haciendo que sean atravesados como si nunca hubiesen estado ahí. Podrá volver a materializarlos cuando quiera, matando a todo aquél que esté dentro de un bloque en ese momento.

Cuando los bloques se hacen inmateriales perderán todas sus propiedades físicas, no sólo dejarán de ser tangibles, sino que perderán su peso y su velocidad de movimiento. Una vez se vuelvan materiales de nuevo, comenzarán a acelerarse como deban.

Habrá dos tipos de bloques diferentes: los *bloques metal* y los *bloques nube*. Los primeros serán más potentes, saldrán a más velocidad y tendrán peso, están ideados para usarse de forma más agresiva, para eliminar a los enemigos. Los otros bloques frenarán poco después de ser disparados, quedando suspendidos en el aire; también podrán ser usados para matar, pero su utilidad es más estratégica.

META

Es el objetivo a alcanzar por el jugador y estará representada por una puerta.

ENEMIGOS

Serán los obstáculos vivientes de cada nivel, podrán moverse libremente por el escenario y cada uno actuará de una forma diferente según sus habilidades (algunos saltarán, otros volarán,

perseguirán, etc.). Morirán al contacto con el jugador, con otro enemigo o con un proyectil.

Cada cierto número de niveles aparecerán enemigos especiales, llamados bosses, que no se podrán sortear; hay que destruirlos para llegar a la meta. Además, no serán tan vulnerables como el resto, en vez de morir al ser disparados, tendrán una vida que irá disminuyendo, cuando ésta se agote, entonces morirán.

JUEGO DE UN JUGADOR

Cada escenario tendrá un límite de bloques de cada tipo y un conjunto de enemigos que se situarán inicialmente en un punto concreto del mapa; conforme el jugador los visualiza por primera vez, éstos comenzarán a actuar como les corresponda. Además, toda criatura que toque a otra morirá, ya sean un enemigo y el jugador, o dos enemigos.

En cada pantalla habrá también una meta alcanzable que servirá para cargar el siguiente mapa una vez el jugador haya llegado a ella.

Habrá un truco que permita avanzar en el juego más rápidamente con la finalidad de probar los escenarios, tener de forma rápida una idea de como quedan, etc. Al teclear el código 'chuck' (sin comillas) el jugador se hará invencible y podrá saltar en el aire. La forma para cancelarlo y volver al estado inicial será mediante el suicidio.

JUEGO EN RED COOPERATIVO

Al jugar en red los jugadores compartirán los bloques (tanto la munición como la capacidad para inmaterializarlos) y podrán dispararse entre ellos; aunque no harán colisión, por tanto, podrán atravesarse sin morir en el intento. Se deberá esperar a que todos los jugadores mueran para reiniciar el mapa, y bastará con que uno llegue a la meta para que todos pasen al siguiente nivel.

Habrá un sistema de puntuaciones: el primero que llegue a la meta, ganará puntos; si alguien muere, los perderá.

JUEGO EN RED COMPETITIVO

Es muy similar al enterior, con la diferencia de que no habrá ni enemigos ni meta; cuando quede un jugador vivo se le darán a éste los puntos (como si hubiera llegado a la meta) y se reiniciará la partida.

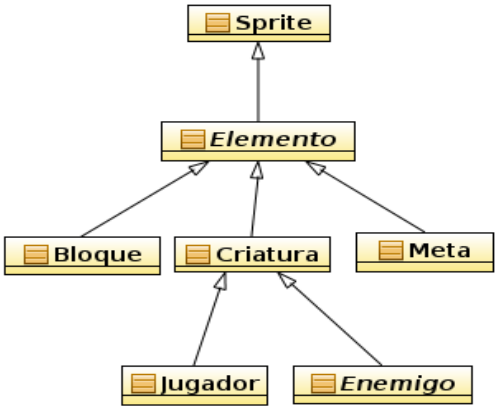


Diagrama de clases genérico de los elementos que aparecen en el juego.

Como clase más básica del juego se utilizará Sprite, que viene proporcionada por la librería Brackeen. Ésta se completará con la clase Elemento para satisfacer posibles requerimientos específicos del juego. Los bloques son elementos simples que pueden materializarse-inmaterializarse y las criaturás serán máquinas de estados. Los posibles enemigos realizarán diferentes acciones según su IA.

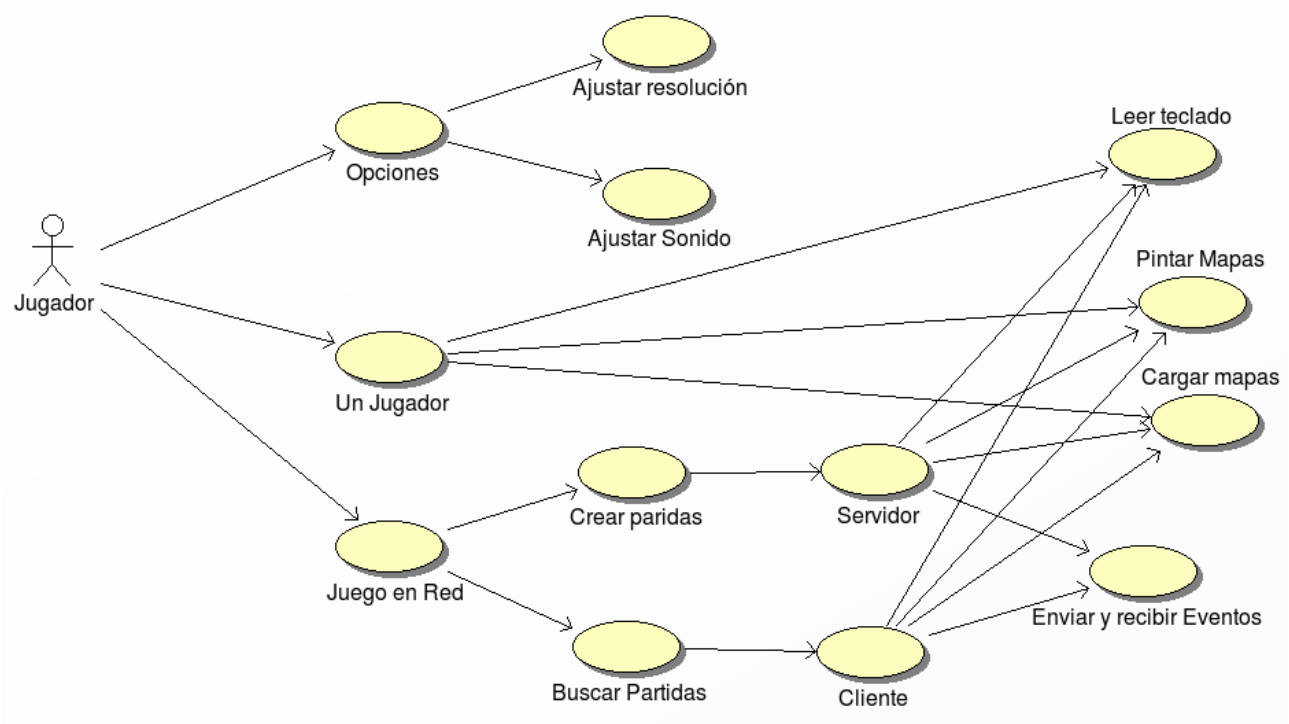
3.1.3 - Esquema general de la aplicación: Casos de Uso

El juego comenzará con un menú principal, en el que se puede elegir jugar uno solo, jugar en red (crear y unirse a partidas), ajustar las opciones (resolución y volumen) o salir.

En todo momento sonará una música de fondo.

En los casos en los que se esté jugando se deberán poder cargar y pintar los mapas así como leer las pulsaciones de teclado para mover a los personajes.

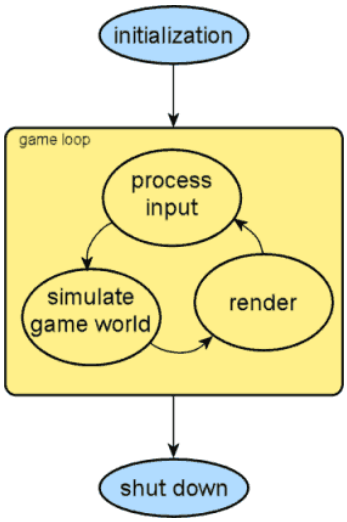
En el caso del cliente y el servidor, éstos deben ser capaces de comunicarse mediante sockets utilizando un protocolo en el que se informe del estado actual del juego.



3.2 - Descripción inicial del Núcleo del Juego

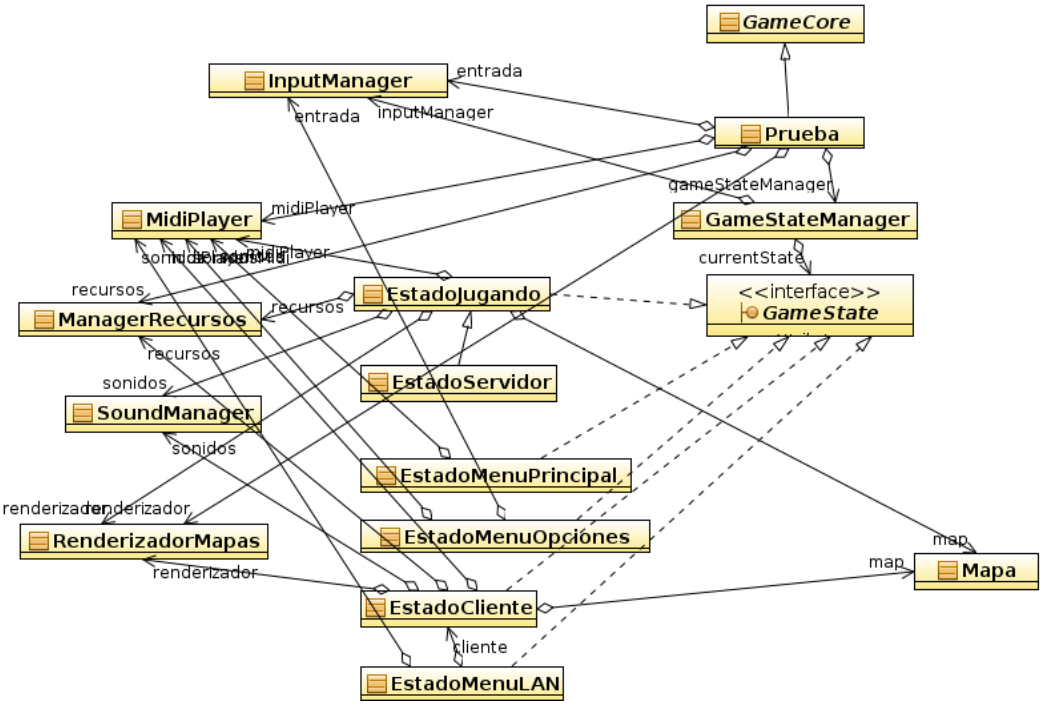
Para la realización de este proyecto se ha hecho uso de una librería orientada a objetos que simplifica tanto el diseño como la implementación del videojuego:

La clase principal, que se ejecuta, tiene dos funciones fundamentales; en primer lugar inicia las demás clases, instanciando, cargando recursos, etc. y posteriormente se encarga de hacer un bucle (o gameloop, similar en todos los videojuegos) en el que lee la entrada, actualiza el juego, y finalmente muestra por pantalla el resultado. Las actualizaciones de la partida se harán de forma constante, así como la frecuencia a la que se muestra por pantalla (los fotogramas por segundo).



Esquema del bucle principal o gameloop.

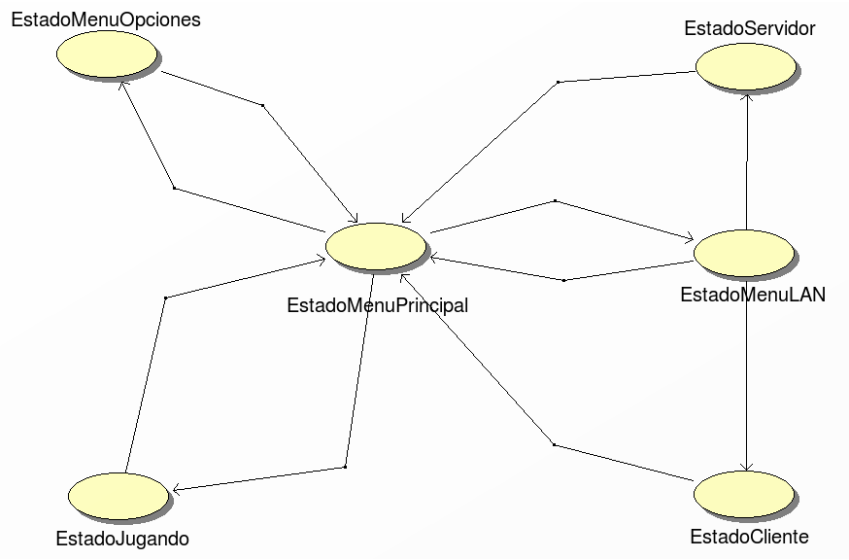
Hay varios tipos de clases según sus fines: las que se encargan de los periféricos (teclado, pantalla y sonido), las clases que se encargan de leer el disco duro (para cargar imágenes y sonidos, por ejemplo), y las clases que representan los estados del juego (los menús, el cliente, el servidor, etc.). Hay un gestor de estados que se encarga de actualizar y pintar los estados que corresponda, así como cambiar el estado en el que nos encontramos.



La clase ejecutable Prueba instancia a las demás; las clases InputManager y MidiPlayer se encargan del teclado y la música respectivamente, mientras que SoundManager se encarga de reproducir sonidos de disparos, gritos, etc.; ManagerRecursos carga del disco las imágenes, sonidos, canciones y mapas necesarios; cada clase Estado* se dedica a una cosa concreta del juego (la lógica del juego, mostrar botones para cambiar de menú, crear y buscar partidas, etc.) y GameStateManager las gestiona a todas ellas. La clase Mapa guarda en memoria el escenario que ha cargado el ManagerRecursos.

Diagrama de transición de estados

En el siguiente diagrama se puede observar los diferentes estados en los que se puede encontrar el juego, para cambiar de uno a otro se usarán teclas y botones (en el caso de los menús). Cada estado indica qué se debe dibujar, qué se debe actualizar y a qué estados se puede cambiar.



3.3 - Diseño de Niveles

A continuación se presenta el diseño de dos escenarios, cómo serán y cómo deberán resolverse (aunque puede haber más formas):

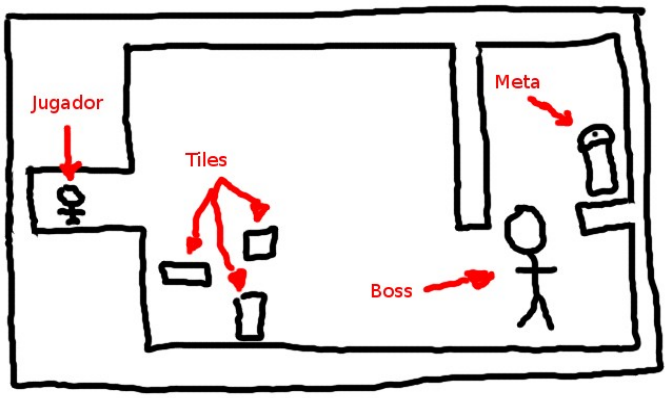
Escenario con Boss (Jefe)

PLANTEAMIENTO

El nivel se presenta como un tunel al principio del cual se encuentra el jugador. Una vez éste avance, habrá una zona más despejada con algunos tiles suspendidos o formando pequeñas paredes; al otro lado del habitáculo se encuentra el Boss, un Ninja. Detrás de él, a cierta altura, está la meta. Para que el jugador no pueda sortear al enemigo y alcanzar el objetivo sin matarlo, hay una pared que va desde el techo hasta tapar parte del Boss.

El comportamiento del contrincante será bastante sencillo, cada cierto tiempo disparará, desde un punto concreto, varios proyectiles (Shurikens) con velocidades aleatorias que rebotan por la pantalla durante un rato o hasta chocar con alguien (entre ellos, con el jugador o con el propio Boss).

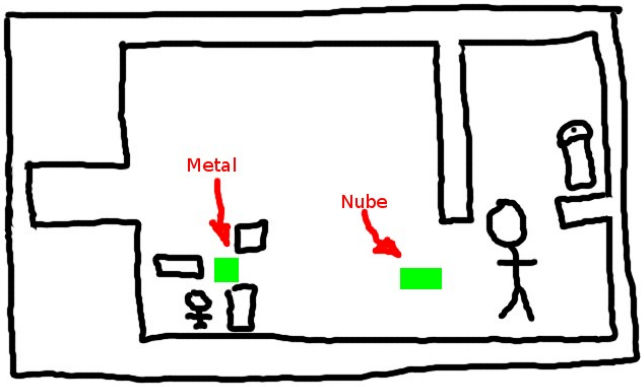
La munición para el nivel será una única nube y muchos bloques (unos 20).



Esquema con el planteamiento inicial.

SOLUCIÓN

En un primer momento, el jugador intentará matar al Boss utilizando los numerosos bloques de metal, guardándose la nube para alcanzar posteriormente la meta. Sin embargo, esta opción no es buena, dado que el Boss recibe poco daño y hay que estar esquivando continuamente sus proyectiles. La mejor solución es aprovechar las características del propio enemigo; en este caso, al saber que los Shurikens los arroja siempre desde el mismo punto y a él también le afectan, bastará con disparar la nube en el punto exacto para que los proyectiles le reboten al emerger. Una vez hecho esto, sólo hay que esperar a que él mismo se destruya. Por si acaso alguno de los proyectiles consigue evitar la nube, el jugador puede aprovechar el escenario para hacer un bunker y meterse dentro. Finalmente, cuando el Boss haya sido derrotado, el camino para llegar a la meta se construirá apilando los bloques de metal.



Esquema con la solución:
el jugador ha creado un bunker y el Boss se destruye a sí mismo.

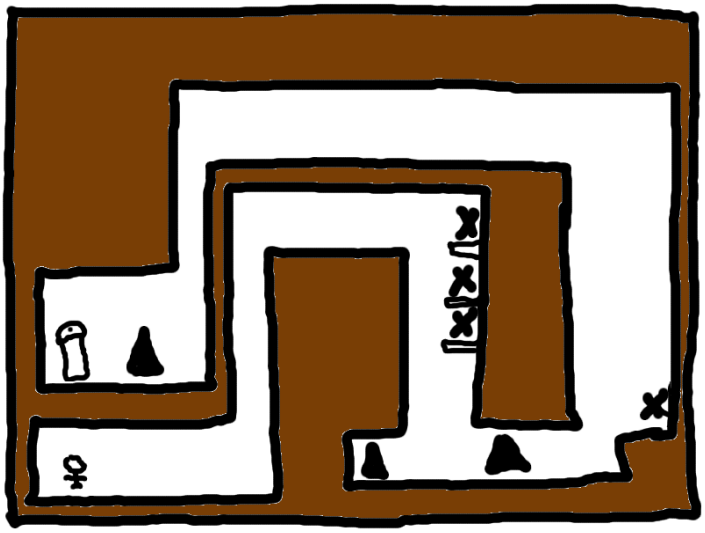
Escenario Laberíntico

PLANTEAMIENTO

Al comenzar el nivel, el jugador se encontrará en la parte inferior del escenario con la meta a la vista y un enemigo protegiéndola, pero el único acceso a la misma es un camino situado en la parte superior, por lo que solo se puede llegar dando la vuelta completa al mapa. Lo único que se puede hacer es avanzar a unas escaleras y subir hasta una plataforma, desde ahí hay que tirarse al vacío por el otro lado, donde esperan dos enemigos para cazar al jugador. Además, en la bajada hay otros enemigos disparando. Finalmente hay, en el extremo más oriental, un último individuo disparando.

La munición para el nivel será un único bloque de metal y varias nubes.

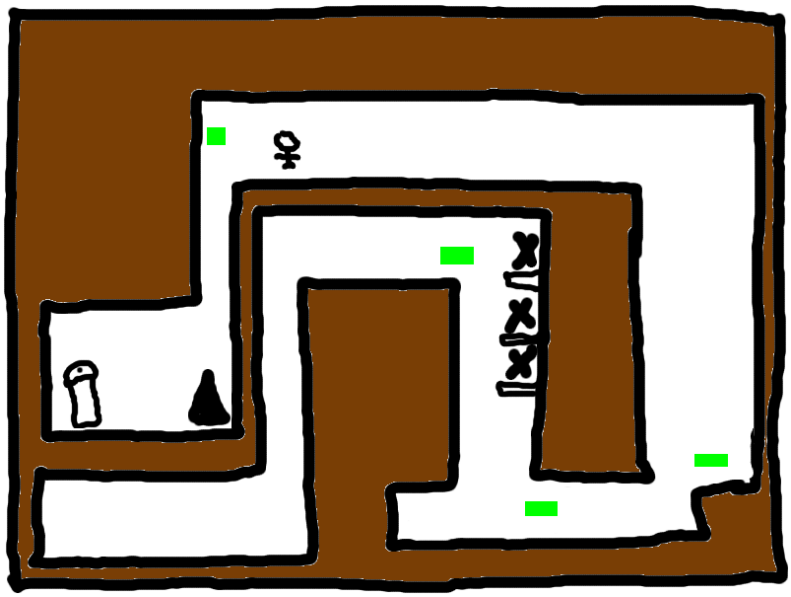
Se presenta un esquema de la situación inicial. Los triángulos representan enemigos que persiguen al jugador cuando éste está cerca, y las cruces son los enemigos que disparan (lo hacen cada cierto tiempo). La parte pintada de marrón es pared, terreno inaccesible.



Esquema inicial.

SOLUCIÓN

Este nivel está planteado para que el usuario lo supere mediante el sistema de prueba y error. El jugador debe subir al montículo y poner una nube lo más a la derecha que pueda, de este modo, el enemigo que tiene delante no puede alcanzarlo, ya que sus proyectiles pegarán en la nube. En segundo lugar, hay que esperar el momento oportuno para lanzarse sin ser alcanzado por los disparos, inmaterializando la nube anteriormente colocada para poder atravesarla. Una vez abajo hay varias opciones, la pensada requiere un poco de velocidad y destreza; consiste en disparar una nube al enemigo que espera a la derecha y una vez lo ha matado, inmaterializarla. Acto seguido, hay que atravesarla, y cuando el individuo de la izquierda esté parcialmente dentro de la nube al perseguir al jugador, se debe volver a materializar, matándolo con la acción. A partir de aquí todo es bastante sencillo, se mata al enemigo de la derecha del todo con otra nube y se sube hasta arriba (con escaleras o mediante nubes, aún no se ha pensado en eso). Una vez arriba falta hacer una última cosa antes de llegar a la meta; si el jugador es perspicaz se dará cuenta de que el primero de los enemigos que aparece, al lado de la meta, se ha quedado debajo del agujero por el que se debe bajar. Esto es así porque al subir el bebé a la plataforma, aquél ha intentado perseguirle, quedándose pegado a la pared. Hay que matarlo antes de tirarse, aquí es donde hace falta el bloque de metal: en primer lugar hay que dejar los bloques inmaterializados, para que pierdan velocidad y peso; luego hay que acercarse al agujero y disparar el bloque en cuestión; por último se vuelve a materializar y la gravedad hará el resto. Este último paso se puede hacer sin inmaterializar los bloques, pero eso requiere mucha precisión, pues el agujero es estrecho y el proyectil puede rebotar en la pared sin llegar a caer, quedándose en el piso de arriba.



Esquema de como debe acabar el nivel durante el último paso, al materializar los bloques, el de metal caerá sobre el enemigo.

4 – IMPLEMENTACIÓN

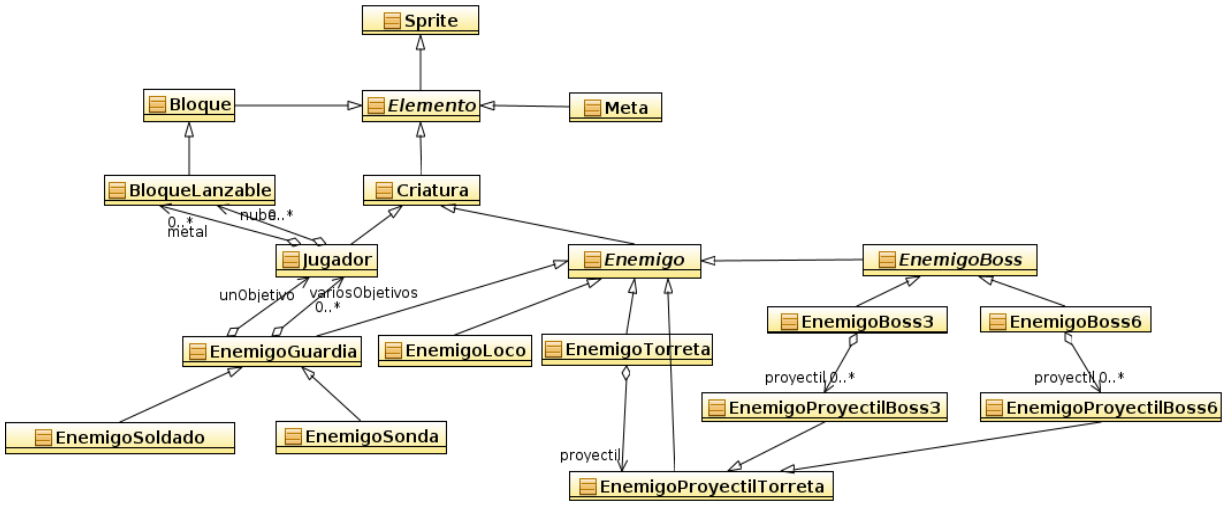
La implementación del proyecto se ha hecho en Java, utilizando como apoyo una librería libre y accesible desde la red: la librería Brackeen. Se ha elegido este lenguaje por varias razones: en primer lugar es de alto nivel y orientado a objetos, simplificando su uso, más intuitivo y sencillo; en segundo lugar, Java posee una extensa y muy bien documentada API, lo que facilita las tareas tanto a la hora de hacer interfaces, como al implementar los Sockets, etc; además, Java es multiplataforma, lo que permite poder llevar el juego de un sistema operativo a otro sin escribir más código del necesario; finalmente, este lenguaje es el único orientado a objetos que se estudia en la carrera, por tanto es el más trabajado y en el que más experiencia se tiene.

4.1 - Implementación de las Clases

Este apartado se subdivide a su vez en dos subapartados, el primero explicando cómo se ha implementado la parte que concierne a los elementos del juego y sus comportamientos; mientras que el segundo explica todo lo relacionado con el núcleo del juego, las interfaces, sonido, etc.

4.1.1 - Elementos del Juego

Los elementos del juego son todos aquellos que pueden representarse en el universo del juego e interaccionar con el jugador de forma visible. Este es el diagrama de clases:



La clase primordial, de la que heredan todas las demás, es la clase Sprite. Ésta, que está incluida en la librería de apoyo, contiene todo lo necesario para representar de forma simple un elemento en el universo del juego: tiene una posición en el espacio bidimensional, una velocidad y una animación (clase de la misma librería que se compone de una secuencia de imágenes).

La clase Elemento se ha hecho con el fin de complementar y ampliar la clase anterior, la nueva clase contiene además información sobre el peso del objeto, la dirección a la que está mirando, y un rectángulo utilizado para las colisiones, de este modo se puede hacer que un elemento pueda superponerse a otro gráficamente sin que ocurra nada. Esto es muy útil para la estética del videojuego.

Como se puede observar en el diagrama, hay tres tipos de elementos diferentes: los bloques, elementos en un principio estáticos que acaban formando parte del escenario; las criaturas, elementos dinámicos que se mueven e interaccionan de forma más activa con el jugador; y la meta, un elemento especial que sirve para indicar cuándo ha conseguido el jugador acabar un nivel.

Los bloques lanzables son un tipo de bloques con un comportamiento específico, con el fin de poder ser disparados por el jugador, e incluso poder inmaterializarse. Como todos se inmaterializan a la vez, se han utilizado métodos y variables estáticas con ése fin. Se han separado las clases Bloque y BloqueLanzable con la idea de hacer otro tipo de bloques con comportamientos diferentes, aunque por ahora no se ha hecho nada.

Diferentes tipos de bloques.

La clase Meta es un elemento cuya única interacción con el mundo es la colisión con un jugador; cuando esto ocurre, se carga un nuevo mapa.

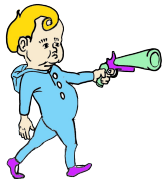
Imagen que representa la meta.

La clase Criatura es en realidad una máquina de estados en la que cada instancia representa un elemento destructible con la capacidad de moverse debido a algún tipo de inteligencia. Todas las criaturas tienen tres estados: vivo, muriendo y muerto, que sirven para determinar su comportamiento posterior y la animación a mostrar. Cabe decir que todas las criaturas, a excepción de los Bosses, mueren al chocar con otra criatura o con un bloque que se esté moviendo (a excepción de dos jugadores).

Para no sobrecargar el ordenador de cálculos innecesarios, las criaturas empiezan inactivas, es decir, ni se actualizan ni se pintan, pero en el momento en que una criatura aparece en pantalla, ésta se despierta. Ésto sólo sucede en modo un jugador, en modo multijugador, tras cargarse el mapa todas las criaturas son despertadas.

Hay dos tipos de criaturas, la primera es la clase Jugador, su comportamiento viene dado por la pulsación de las teclas, pude saltar, suicidarse y disparar de forma limitada diferentes tipos de bloques e inmaterializarlos a voluntad. Mientras está vivo, tiene una serie de sub-estados, que son quieto, andando y saltando; estos estados sólamente sirven para hacer una animación u otra.

El jugador puede ser objeto de un truco en el que se convierte en Chuck Norris, apareciéndole un sombrero, haciéndolo indestructible y permitiéndole saltar en el aire.



Personaje Principal.

El otro tipo de criatura es la clase Enemigo, una clase abstracta con el método *hacerIA()*, en el que cada clase que hereda de ella hará un comportamiento concreto.

El enemigo más simple es EnemigoLoco, cuyo comportamiento es moverse horizontalmente hasta encontrar un obstáculo y dar media vuelta.

Enemigo Loco.

EnemigoGuardia es una versión mejorada del Loco, tiene una lista con todos los jugadores, de este modo sabe en cada momento dónde está el jugador más cercano y le persigue si se acerca demasiado.

Imagen de EnemigoGuardia.

EnemigoSoldado y EnemigoSonda extienden de EnemigoGuardia, el primero es como el guardia, solo que puede saltar los obstáculos del camino del siguiente modo: guarda la última posición en la que ha estado, si está persiguiendo a un jugador y su posición actual es la misma que la anterior significa que hay un obstáculo que le impide avanzar, de modo que, al igual que el jugador, salta (utilizando una velocidad vertical concreta).

EnemigoSonda puede volar, pudiendo atacar al jugador desde arriba.

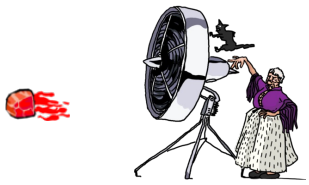
Enemigo Sonda.

La clase EnemigoProyectilTorreta también es simple, el objeto avanza lateralmente hasta una distancia máxima y muere al alcanzarla.

La clase EnemigoTorreta se dedica a disparar instancias de la clase anterior, una vez ha muerto el proyectil, lo vuelve a disparar.

Torreta con su proyectil.

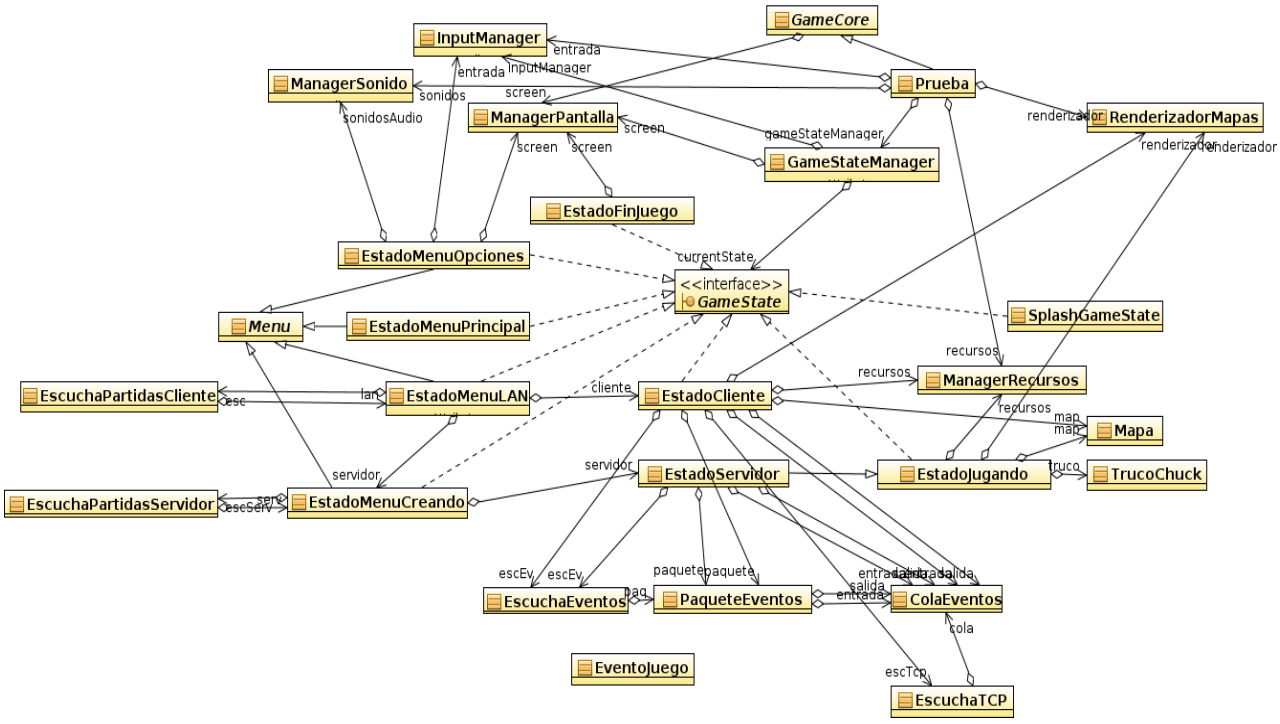
EnemigoBoss es una clase abstracta especial, en vez de morir con un impacto, va perdiendo vida, cuando ésta llega a cero, entonces muere. Cada Boss dispara cada cierto tiempo un número de proyectiles concreto, cada tipo de proyectil tiene su propio comportamiento.



Uno de los boss con su proyectil.

4.1.2 - Núcleo del Juego

Este apartado contiene todo lo correspondiente al funcionamiento del juego: navegación entre menús, lógica y colisiones de la partida, conexiones en red, sonido y resolución, etc. Este es el diagrama de clases:



GameCore

Como ya se había explicado en el diseño, existe un bucle infinito en la aplicación que se encarga de leer las entradas del usuario, actualizar el juego según esas entradas y mostrar los cambios por pantalla. Cada estado del juego se encarga de leer las entradas necesarias, mientras que GameCore implementa el resto del núcleo, llamando al método draw() para dibujar y al método update() para actualizar según el tiempo que le haya costado la última iteración (llamado elapsed time).

```
init();
While (isRunning) {
        elapsedTime = calcularTiempoTranscurrido();
        //actualizar
        update(elapsedTime);
        //pintar
        Graphics2D g = screen.getGraphics();
        draw(g);
        g.dispose();
        //breve pausa
        Thread.sleep(12);
}
```

Se hace una inicialización de parámetros con el método init() y luego se ejecuta el bucle de forma indefinida; el sleep se hace porque de otro modo el manager de entrada (inputManager) no detecta bien las pulsaciones.

Dibujar objetos en Java puede ser costoso, es por ello que surgen dos problemas: en primer lugar, si se pinta cada iteración del bucle, la máquina virtual consume más recursos de los necesarios, ya que el ojo humano no es capaz de diferenciar las imágenes a tanta frecuencia. Para que el juego se vea fluido basta dibujar 50-60 veces por segundo, haciendo cálculos se obtiene que con pintar aproximadamente cada 16 milisegundos es suficiente. El código cambiado de GameCore:

```
tiempoDibujo+=elapsedTime;
if (tiempoDibujo>=TIEMPO_DRAW) {
        tiempoDibujo-=TIEMPO_DRAW;
        Graphics2D g = screen.getGraphics();
        draw(g);
        g.dispose();
}
```

donde TIEMPO_DRAW indica cada cuánto se pinta (16 milisegundos), elapsedTime contiene el tiempo que ha costado la iteración y tiempoDibujo es un contador del tiempo que pasa sin pintar.

En segundo lugar, no tiene por qué transcrrir el mismo tiempo en cada iteración, siempre hay unas pequeñas diferencias que hacen que el tiempo de actualización varíe; de este modo, sucesos

tales como disparar o andar no tienen dos veces el mismo resultado, es decir, dos bloques disparados desde el mismo sitio no llegan al mismo lugar. Para solucionarlo, lo que se hace es dividir el tiempo de actualización en bloques de 5 milisegundos y actualizar tantas veces como lo permita el tiempo transcurrido en la iteración. Este es el código ajustado:

```
tiempoPasado+=elapsedTime;
while(tiempoPasado>=TIEMPO_UPDATE) {
        update(TIEMPO_UPDATE);
        tiempoPasado-=TIEMPO_UPDATE;
}
```

Prueba

Esta es la clase principal, la que contiene el método Main() y se ejecuta. Extiende de GameCore.

En primer lugar, se encarga de detectar y almacenar en qué sistema operativo se ejecuta la aplicación con el método *System.getProperty("os.name")*, ya que MAC puede dar problemas al dibujar con el swing de java o al intentar usar cursores personalizados; en segundo lugar, se encarga de instanciar el resto de clases necesarias (sonidos, entrada, recursos, renderizador, gameStateManager, etc.) y de introducir cada estado de juego en el gameStateManager:

```
gameStateManager.addState(
                    new SplashGameState("imagenCarga.png",screen,midiPlayer,sonidos));
gameStateManager.addState(new EstadoJugando(sonidos,midiPlayer,screen));
gameStateManager.addState(new EstadoMenuPrincipal(sonidos,midiPlayer,screen));
gameStateManager.addState(
                  new EstadoMenuOpciones(sonidos,recursos,midiPlayer,screen,entrada));
gameStateManager.addState(new EstadoFinJuego(screen,midiPlayer));
gameStateManager.addState(new EstadoMenuAyuda(midiPlayer,screen));
```

Finalmente, se cargan todos los recursos (imágenes, sonidos, etc.) y se establece el estado de juego inicial en un hilo aparte:

```
new Thread() {
        @Override
        public void run() {
                gameStateManager.loadAllResources(recursos);
                // evita que los hilos de pintar interfaces interaccionen entre sí
                NullRepaintManager.install();
                gameStateManager.setState(GameStateManager.SPLASH);
        }
}.start();
```

GameStateManager

Esta clase es simplemente un gestor de estados de juego que están contenidos en una lista. Se pueden añadir estados con el método *addState()* y establecer un estado concreto con *setState()*, como se observa en el código mostrado de la clase anterior. Con los métodos *update()* y *draw()* se actualiza y pinta el estado seleccionado como actual, mientras que el método *isDone()* se utiliza para saber si el juego debe acabarse y detenerse el bucle. Para esto hay un estado especial sin instancia, llamado EXIT_GAME, que cuando se alcanza hace detener el bucle:

```
public void setState(String name) {
    // clean up old state
    if (currentState != null) {
       currentState.stop();
    }
    inputManager.clearAllMaps();

    if (name.equals(EXIT_GAME)) {
       done = true;
    }
    else {
       // set new state
       currentState = (GameState)gameStates.get(name);
       if (currentState != null) {
          currentState.start(inputManager);
       }
    }
}
```

CLASES DE COMUNICACIÓN EN RED

EventoJuego

Ésta es la clase básica de comunicación, sirve para enviar mensajes entre el cliente y el servidor tanto en TCP (implementa la clase serializable) como en UDP. Contiene una serie de variables que indican qué tipo de evento es (alguien se ha conectado, alguien ha disparado, una criatura ha muerto, etc.), quién lo ha producido y qué cambios hay que realizar; dependiendo del tipo de evento se usan más o menos variables con una u otra finalidad. Por ejemplo, si el servidor quiere mandar las puntuaciones de los jugadores, establece el tipo de evento S_PUNTUACION, introduce el identificador del jugador y escribe en la variable *nuevaX* los puntos de ese jugador; el cliente, al recibir el evento y ver que se trata de ese tipo, ya sabe qué variables leer y qué hacer con ellas. En otro caso, si por ejemplo el servidor quiere mandar a los clientes información sobre la munición que queda, se establece el tipo S_PONER_BLOQUES, y se escribe en *nuevaX* y en *nuevaY* la cantidad restante de bloques de metal y de nubes. Como se observa, se han utilizado diferente número de variables y con diferentes funciones según el tipo de evento.

La interfaz ProtocoloEventos contiene todas las constantes que representan los tipos de eventos del juego. Hay 21 clases de mensajes diferentes y están divididos en tres grupos, los

orientados al diálogo de la conexión, los enviados por el cliente durante el juego y los enviados por el servidor también durante la partida.

PaqueteEventos

Esta clase prepara una serie de eventos de juego para enviarlos por UDP como un array de bytes; también es capaz de lo inverso, es decir, de obtener eventos de juego a partir de un array de bytes. Como su estructura se explica más adelante, bastará aquí con mencionar que utilizando su método *empaquetar()* construye un array de bytes, a partir de un número máximo de eventos contenidos en una cola, con el fin de utilizarla como contenido de un datagrama. Con el método *desempaquetar()* transforma un array de bytes proveniente de un datagrama en un conjunto de eventos de juego que introducirá en una cola de eventos.

ColaEventos

ColaEventos es un listado o cola dinámica que contiene objetos de la clase EventoJuego. Se utilizan por parejas tanto en el servidor como en el cliente para contener los eventos entrantes y salientes.

EscuchaEventos

La clase EscuchaEventos extiende de Thread, o lo que es lo mismo, es un hilo que se ejecuta en paralelo; su función es estar de forma indefinida escuchando paquetes UDP en el puerto correspondiente, y una vez llega uno, desempaquetarlo y encolar su contenido.

```
byte buff[]=new byte[5000];
while (!acaba) {
        DatagramPacket p=new DatagramPacket(buff,buff.length);
        try {
                sock.receive(p);
                paq.desempaquetar(p.getData());
        }
        catch (Exception e){
                System.out.println("error escucha eventos: "+e.toString());
        }
}
```

El bucle se ejecuta continuamente hasta que el método *acabar()* es llamado, momento en el que la condición del *while* deja de cumplirse y el hilo finaliza. Esto se hace en los métodos *stop()* de las clases EstadoServidor y EstadoCliente

EscuchaTCP

Esta clase tiene la misma estructura y funcionalidad que la anterior, con la diferencia de que está pendiente de los mensajes enviados por TCP, estos eventos son transmitidos así debido a que solo se generan una vez, y deben llegar con seguridad al destino. Son los eventos para cargar un mapa (el mismo o el siguiente), si este mensaje no llegara, las nuevas posiciones de los elementos no se corresponderían, pudiendo aparecer los personajes flotando, los enemigos atravesando paredes, etc. El otro tipo de evento es el de desconexión, si no se recibe por parte de algún cliente el juego simplemente deja de responder (ya no hay servidor que envíe los cambios) y el proceso debe ser matado a la fuerza.

EscuchaPartidasCliente

EscuchaPartidasCliente es una clase utilizada por EstadoMenuLAN que implementa Runnable, esto es, se puede ejecutar en paralelo mediante un hilo. Se encarga de buscar partidas que han sido creadas por un servidor y que se están anunciando a broadcast, así como de estar pendiente de cuando empieza la partida o de si el servidor ha decidido cancelarla. Para buscar las partidas, el hilo está, durante un tiempo configurable (5 segundos), escuchando en un socket UDP los eventos de juego; cuando llega uno, si es diferente de S_PARTIDA_CREADA lo descarta, si no, coge la información contenida y la incluye en una lista mediante el método *añadirPartida()* del menú LAN, que posteriormente las mostrará mediante botones.

Para comenzar a jugar a la vez que el servidor, una vez que el cliente se une a la partida se vuelve a ejecutar el hilo, pero esta vez espera eventos del tipo S_COMIENZA_PARTIDA o S_DESCONEXION.

EscuchaPartidasServidor

Esta clase es usada por EstadoMenuCreando de forma similar a la anterior: implementa la clase Runnable y se encarga de escuchar eventos de juego. La diferencia es que EscuchaPartidasServidor tiene un ServerSocket que acepta y rechaza conexiones según los nombres de los jugadores. El funcionamiento conjunto de estas clases se explica más adelante, en la imlpementación del protocolo de red.

ESTADOS DEL JUEGO

Interfaz GameState

Todos los estados del juego deben implementar la interfaz GameState con los siguientes métodos: *getName()*, para saber el nombre del estado; *checkForStateChange()*, para saber si hay que cambiar de estado y a cuál hay que cambiar; *loadResources()*, para cargar todos los elementos al inicio del juego, ya sean los sonidos en el caso de los estados de juego o la interfaz y los botones

en el caso de los estados de menús; *start()*, para iniciar el estado: cargar el mapa inicial, establecer las teclas, iniciar la música, etc; *stop()*, para finalizar el estado: parar los sonidos, cerrar sockets, etc; *update()*, para actualizar el estado; y *draw()*, para dibujarlo.

Los estados se pueden agrupar en dos tipos diferentes, los menús, que son básicamente una interfaz para elegir las opciones y modos de juego; y los estados del juego en sí, que son los que cargan, actualizan y dibujan los mapas. Hay otros estados más independientes, como SplashGameState, que se inicia al arrancar el progarama y muestra una imagen mientras se cargan todos los recursos; o EstadoFinJuego, que anuncia al jugador que ha superado todos los niveles del juego.

SplashGameState

Éste es el estado inicial, muestra una imagen por pantalla durante unos segundos, mientras cargan los recursos del juego en otro hilo. Cuando ha pasado cierto tiempo, se establece MENU_PRINCIPAL como siguiente estado y ya no se vuelve a acceder a SplashGameState.

EstadoFinJuego

Similar a la clase anterior, muestra por pantalla un mensaje de felicitación una vez que todos los niveles han sido superados. Una vez se pulsa una tecla, vuelve al menú principal.

Menu

Esta clase abstracta sirve para poner en común varios métodos a los diferentes estados de tipo menú. Implementa el método *crearBoton()*, que devuelve un JButton con unos iconos a partir de varias imágenes; y el método *crearCursor()*, que crea cursores personalizados (aunque en MAC hay problemas para hacerlo, por lo que en ese caso devuelve el cursor por defecto). La clase Menu tiene también un método abstracto, *actionPerformed()*, para gestionar los eventos de pulsaciones de botones.

Todos los estados que extienden de Menu tienen botones emparejados con teclas, de modo que al pulsar una determinada, se simula la pulsación del botón correspondiente. También constan de un ContenPane (clase en la que se ponen los objetos de Swing para que se dibujen) personalizado que permite poner imágenes y animaciones como fondo de los menús.

EstadoMenuPrincipal

El menú principal es una clase muy simple, consta de varios botones para cambiar de estado: se puede comenzar una partida de un jugador, ir al menú de LAN, acceder al menú de opciones y salir. Más adelante se le pueden añadir otras funcionalidades, como acceder a un menú de ayuda, ver los créditos, etc.

EstadoMenuOpciones

Esta clase es más compleja que las anteriores, en ella se pueden cambiar tanto la resolución de pantalla como el volumen de la música. Para cambiar la resolución, en primer lugar se registra cuál es la que tiene por defecto la pantalla (pantalla completa) y se crea una lista con otras resoluciones; cuando el usuario elige una, se crea un nuevo JFrame del tamaño indiacado, si la resolución elegida no es la inicial, se le añaden bordes a la pantalla para convertirla en modo ventana. En el cambio de sonido, se ha establecido una interfaz para cambiar el volumen que admite una entrada entre cero y uno, basta con dividir el punto actual del scrollbar entre su máximo para hallar el nuevo volumen:

```
float nuevoVolumen=((float)(volumen.getValue()))/((float)volumen.getMaximum());
sonidosMidi.setVolumen(nuevoVolumen);
sonidosAudio.setVolumen(nuevoVolumen);
```

EstadoMenuLAN

Este menú permite crear, buscar y unirse a partidas en red. La explicación del funcionamiento en red se explica más adelante; aquí bastará decir que EstadoMenuLAN consta, aparte de los botones correspondientes a búsqueda, creación de partida, etc., de un campo de texto donde introducir el nombre. Esto supone un gran problema, el hilo que pinta los componentes es por lo general el propio bucle del juego, pero en este caso, dado que se pueden introducir datos por teclado, la aplicación puede bloquearse; para solucionarlo, se crea un hilo para pintar swing en el método *start()* y se quita en el método *stop()*. Otro problema añadido es el MAC, en el caso de que la aplicación corra en un ordenador con ese sistema operativo, los hilos de swing no funcionan bien, así que no hay más remedio que pintar en el bucle del juego. Para evitar el bloqueo se inhabilita el campo de texto y el nombre se pide al iniciar la aplicación, antes de crear los componentes. Todo esto se sabe accediendo a un campo de la clase Prueba, que detecta el sistema operativo:

```
public void loadResources() {
        ...
        if (Prueba.problemasMAC) {
                nombre.setText(nombreMAC);
                nombre.setEnabled(false);
                sePinta=true;
        }
        else {
                sePinta=false;
        }
        ...
}



public void start() {
        ...
        if (!Prueba.problemasMAC) {
```

```
                NullRepaintManager.setCurrentManager(new RepaintManager());
        }
        ...
}

public void stop() {
        ...
        if (!Prueba.problemasMAC) {
                NullRepaintManager.install();
        }
        ...
}


public void draw(Graphics2D g) {
        if (sePinta) {
                g.setColor(Color.ORANGE);
                g.fillRect(0, 0, width, height);
                ventana.getLayeredPane().paintComponents(g);
                screen.update();
        }
}
```

EstadoMenuCreando

Éste es un menú muy simple que permite ver quiénes se conectan a una partida creada, comenzar la partida una vez todos están conectados, y también permite al creador cancelarla. Más adelante se explicará en profundidad su funcionamiento de cara a la red.

EstadoJugando

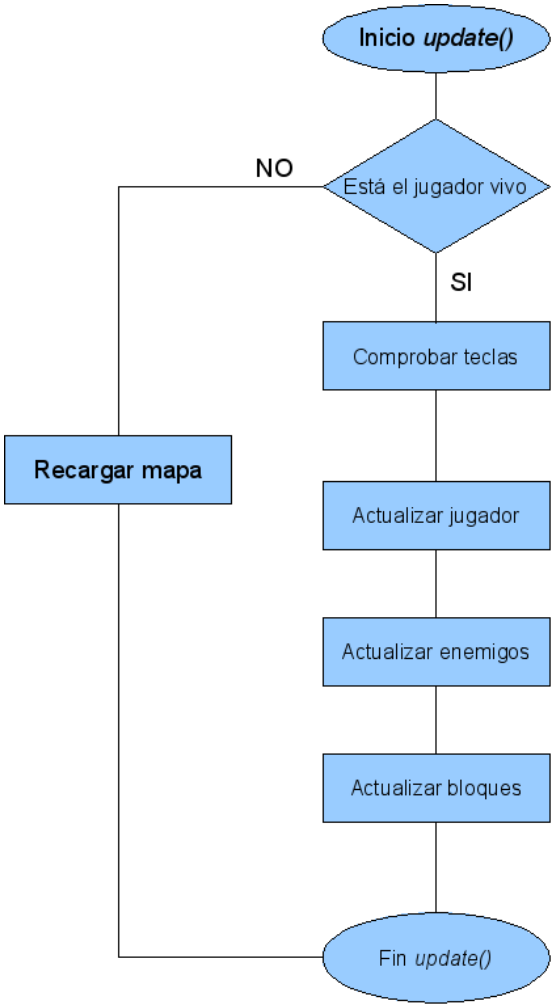
Esta clase es una de las más complejas, en ella se desarrolla toda la lógica del juego, desde la carga de los escenarios hasta la detección de colisiones, así como la ejecución de la inteligencia artificial y la animación de las criaturas. Para pintar mediante el método *draw()* se hace uso de otra clase, RenderizadorMapas, a la cual se le pasan como parámetros el mapa a pintar y las dimensiones. Dado que la resolución puede ser cambiada por el usuario, el procedimiento de dibujo es el siguiente: se dibuja todo como si tuviera la resolución por defecto y luego se escala a la resolución actual mediante la clase AffineTransform:

```
public void draw(Graphics2D g) {
    double tamX=((double)anchoActual)/width;
    double tamY=((double)altoActual)/height;
    g.transform(AffineTransform.getScaleInstance(tamX,tamY));
    renderizador.draw(g, map, width,height);
}
```

donde width y height son respectivamente la anchura y altura originales; por ejemplo, si en una

pantalla de resolución inicial 1280x1024 se ha cambiado el modo de juego a 800x600, anchoActual vale 800, altoActual vale 600, width vale 1280 y height vale 1024.

En cuanto a la parte lógica, contenida en el método *update()*, se puede dividir en varias partes, tal y como se muestra en el siguiente diagrama:



La recarga del mapa se hace utilizando la clase ManagerRecursos, explicada más adelante, mientras que tanto la comprobación de teclas las actualizaciones de los objetos del juego se hacen de forma interna.

La actualización de las criaturas se hace en base al elapsed time (tiempo de una iteración), explicado en clases anteriores, a partir del cual se calcula cuánto debe desplazarse cada una. También en este apartado se hace la detección de colisiones, tanto entre las criaturas como con el escenario. Para todo esto se hacen dos 'updates' diferentes por elemento, en el primero se calcula el cambio de movimiento (colisiones, gravedad, etc.); mientras que el segundo sirve para variar, si la situación lo requiere, el comportamiento del objeto:

En el caso de las criaturas:

```
...
   if (e.estaVivo() && e.estaDespierta()) {
      updateCriatura(e,elapsedTime);
   }
```

```
e.update(elapsedTime);
```

donde la variable *e* es la criatura, con el método *updateCriatura(e,elapsedTime)* se varía la posición y con la llamada a *e.update(elapsedTime)* se modifica la animación, el estado, etc. dependiendo de la criatura en cuestión.

En el caso de los bloques:

```
updateBloque(b,elapsedTime);
b.update(elapsedTime);
```

Cabe decir que tanto los enemigos como los bloques están dispuestos el listas, cuando un enemigo muere es quitado, y cuando un bloque se detiene es llevado a otra lista (ya no se va a mover más); de este modo se consigue actualizar el mínimo número de objetos, optimizando la aplicación.

Ejemplo de su uso:

```
...
else if (e.getEstado()==Criatura.MUERTO) {
        iterador.remove();
}
...

...
if (!bloque.esEtereo() && !bloque.seMueve()) {
        map.pasarAdetenidos(bloque);
}
...
```

EstadoServidor

EstadoServidor extiende de la clase anterior y se encarga de hacer lo mismo que EstadoJugando pero para varios jugadores, recibe eventos de los clientes (mediante las clases EventoJuego, EscuchaEventos y ColaEventos) y actualiza las teclas como si las hubiese pulsado un usuario. Una vez hace todos los cálculos les envia otros eventos a ellos (utilizando la clase PaqueteEventos y de nuevo la clase ColaEventos).

Antes de dar comienzo al juego se debe informar al servidor qué sockets se van a usar, cuántos y quiénes son los clientes, etc. Por ello, exite el método *conectar()*, que es llamado desde el EstadoCreando, para que luego al hacer el método *start()* se puedan poner diferentes hilos a escuchar vía TCP y UDP.

El comportamiento de esta clase también es ligeramente diferente a EstadoJugando debido al hecho de que hay varios modos de juego y de que hay más de un jugador. En primer lugar, la partida no se reinicia hasta que no han muerto todos; en ese caso, para que cada jugador vuelva a

cargar su mapa, se debe enviar un mensaje S_MISMO_MAPA por TCP dado que es único y es de vital importancia que llegue:

```
Jugador[] player = map.getJugadores();
if (todosMuertos(player)) {
        enviarTCP(ProtocoloEventos.S_MISMO_MAPA);
        ...
}
```

En segundo lugar, si se está jugando en modo cooperativo, se debe comprobar si algún jugador ha llegado a la meta; en este caso hay que enviar un mensaje S_NUEVO_MAPA del mismo modo y por la misma razón que se acaba de comentar:

```
...
enviarTCP(ProtocoloEventos.S_NUEVO_MAPA);
...
```

Hay un tercer tipo de evento que debe enviarse por TCP: el de desconexión. Si se enviara por UDP y alguien no lo recibiera, el juego se le quedaría trabado (el servidor ya no le envía cambios ni escucha sus teclas).

También hay, en modo multijugador, un sistema de puntuaciones que debe ser enviado con el resto de información (posiciones y estados de criaturas y bloques, etc.). Ésto se hace mediante UDP y sólo cada cierto tiempo, pues si no se puede sobrecargar la red demasiado.

```
transcurrido+=elapsedTime;
if (transcurrido>=TIEMPO_ENVIO) {
        transcurrido=0;
        enviarCambios();
}
```

donde *transcurrido* es una variable para determinar el tiempo que ha pasado desde el último envío y enviarCambios() es un método que va creando y encolando eventos mediante las clases EventoJuego y ColaEventos, como se ha mencionado anteriormente.

Finalmente, a la hora de pintar EstadoServidor llama al método *pintarLAN()* de RenderizadorMapas, y al igual que EstadoJugando, redimensiona la imagen con AffineTransform para la resolución.

EstadoCliente

Esta clase funciona de forma similar a la anterior, solo que la actualización de los objetos la hace a través de los eventos recibidos, no mediante cálculos.

El envío de paquetes se hace cada cierto tiempo, para no saturar la red; así mismo, la lectura de eventos enviados por el servidor se realiza estableciendo un límite de paquetes a leer.

Envío:

```
transcurrido+=elapsedTime;
if (transcurrido>=TIEMPO_ENVIO) {
   transcurrido=0;
   checkInput(elapsedTime);
   ...
   sock.send(p);
   ...
}
```

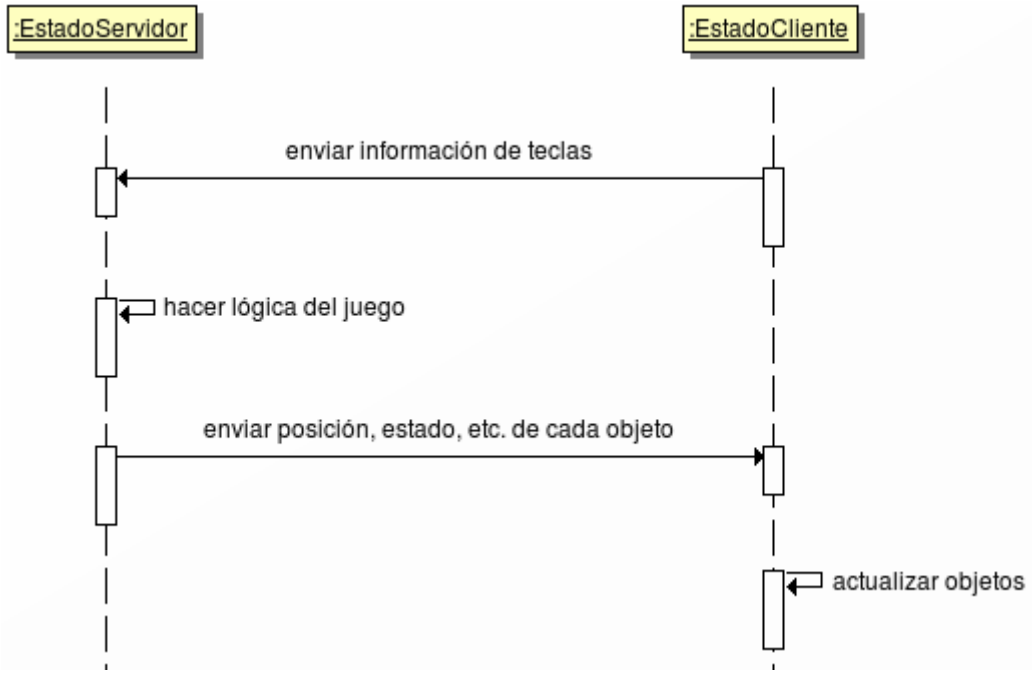
al igual que ocurre en el servidor, cuando ha pasado cierto tiempo (controlado por la variable *transcurrido*) se procesa, en este caso con *checkInput()* y se envían los datos por UDP mediante un objeto *p* de la clase DatagramPacket.

Lectura:

```
EventoJuego e;
int size=Math.min(MAX_PAQUETES_LEIDOS,entrada.size());
for(int i=0;i<size;i++) {
        e=entrada.deQueue();
        tratarEvento(e);
}
```

donde la variable *entrada*, de tipo ColaEventos contiene todos los eventos recibidos, mientras que *size* sirve como tope para leer como mucho un número concreto de eventos.

Un esquema simple del funcionamiento general de la comunicación entre cliente y servidor:



La forma específica en que la información es transmitida se explica de forma más detallada más adelante, en la clase EventoJuego.

EstadoMenuAyuda

Este menú muy simple indica, mediante texto, en qué consiste el juego, cuáles son los controles, etc.

EstadoMenuCreditos

De forma similar a la clase anterior, ésta muestra los participantes, los agradecimientos, etc.

CLASES DE RECEPCIÓN Y EMISIÓN DE DATOS

GameAction

Esta clase representa la interacción del usuario con la aplicación, se asocia a una o varias teclas o botones del ratón mediante el InputManager y luego se consulta si la acción se está llevando a cabo mediante el método *isPressed()*. Se puede establecer si se quiere permitir su detección continuada (se puede dejar pulsada) o sólo la pulsación inicial. También se puede simular su pulsación dentro del código mediante métodos como *tap()* o *press()*; esto es útil, por ejemplo, para el servidor en red, le permite pulsar las teclas de los clientes al recibir los mensajes para ello.

InputManager

InputManager se encarga de gestionar los eventos de ratón y teclado, puede asociar éstos a diferentes GameAction mediante el método *mapToKey()* y *mapToMouse()*; permite crear un cursor invisible e incluso se puede establecer un movimiento relativo del ratón para juegos como shooters en primera persona. Aunque la mayoría de estas funcionalidades no harán falta, dado que no se utiliza el ratón, a excepción de los menús.

Ejemplo de utilización:

```
GameAction saltar=new GameAction("saltar",
                                 GameAction.DETECT_INITAL_PRESS_ONLY);
inputManager.mapToKey(saltar, KeyEvent.VK_SPACE);
inputManager.mapToKey(saltar, KeyEvent.VK_UP);
if (saltar.isPressed()) {
        ...
}
```

con este código se consigue asociar la acción de saltar a las teclas 'espacio' y 'flecha arriba', y sólo se detecta la pulsación inicial, es decir, cada vez que se quiera realizar la acción se debe pulsar una de las teclas, no basta con dejarlas pulsadas. Si se ha detectado la pulsación, *isPressed()* devuelve *verdadero* y se ejecuta el código de dentro del *if*.

SoudManager

Esta clase carga y reproduce diferentes sonidos cortos (como explosiones o disparos) basándose para ello en un conjunto de hilos (ThreadPool). Su funcionamiento es el siguiente: se crean un número concreto de hilos capaces de reproducir sonidos y quedan a la espera; al acceder al método *play()* uno de los hilos en espera reproduce lo que deba y al finalizar vuelve a quedar en espera para reproducir de nuevo cuando sea necesario. Si no hay hilos suficientes para reproducir, los sonidos quedan en espera y suenan más tarde. Tiene otras funcionalidades, como la aceptación de diferentes filtros de sonido (eco, sonido 3d, etc.), pero por falta de tiempo no han podido realizarse.

ManagerSonido

El Manager de Sonido extiende de la clase anterior y tiene un método adicional para cambiar el volumen de los sonidos a través del menú de opciones; aunque, por desgracia, aún no se ha conseguido que éste varíe.

MidiPalyer

MidiPlayer sirve para reproducir melodías con extensión .midi, se utiliza para hacer sonar música ambiental, almacenada en variables de la clase Sequence, durante el videojuego mediante una sencilla interfaz: el método *getSequence()* carga los archivos en memoria, *play()* permite reproducirlo con la opción de bucle (cuando se acaba vuelve a empezar), *setPaused()* sirve para detener o proseguir la reproducción; finalmente se le ha añadido un método, *setVolumen()*, para modificar el volumen en el menú de opciones. Para la realización de este método se ha tenido que cambiar un póco el código original: en el constructor se obtienen dos variables, el secuenciador y el sintetizador; el primero es utilizado para la reproducción de los sonidos, mientras que el segundo sirve para variar el volumen. Ambos deben estar asociados mediante la siguiente línea de código:

```
sequencer.getTransmitter().setReceiver(sintetizador.getReceiver());
```

Para cambiar el volumen, lo que hay que hacer es acceder a los canales midi del sintetizador y variar cada uno de forma individual:

```
public void setVolumen(float volumen) {
        MidiChannel[] channels = sintetizador.getChannels();
        for(int i=0;i<channels.length;i++) {
                channels[i].controlChange(VOLUME_CONTROL,(int)(volumen*127));
        }
        volumenActual=volumen;
}
```

el parámetro pasado al método va entre cero y uno, mientras que el volumen de los canales puede alcanzar 128 valores (0 a 127), por eso la multiplicación.

ScreenManager

Esta clase se encarga de gestionar todo lo relacionado con la pantalla para que funcione en ventana completa o FullScreen mediante la utilización de DisplayModes, que tienen información sobre la resolución de pantalla.

ManagerPantalla

ManagerPantalla extiende de ScreenManager, y tiene sus mismas funcionalidades con la diferencia de que puede establecerse la resolución y la decoración de la ventana de manera dinámica en el menú de opciones mediante el método *setFullScreen()*.

ResourceManager

El ResourceManager implementa métodos para cargar en memoria sonidos, midi e imágenes; así como también tiene métodos para crear imágenes transformadas (en espejo horizontal, vertical, etc.)

Mapa

Esta clase guarda los escenarios, ficheros que contienen la información necesaria para representar la acción del juego: el tamaño del mapa y la estructura de los tiles, la posición y el tipo de cada enemigo, el lugar donde empieza el jugador, etc. Se carga en memoria mediante la clase ManagerRecursos, explicada a continuación.

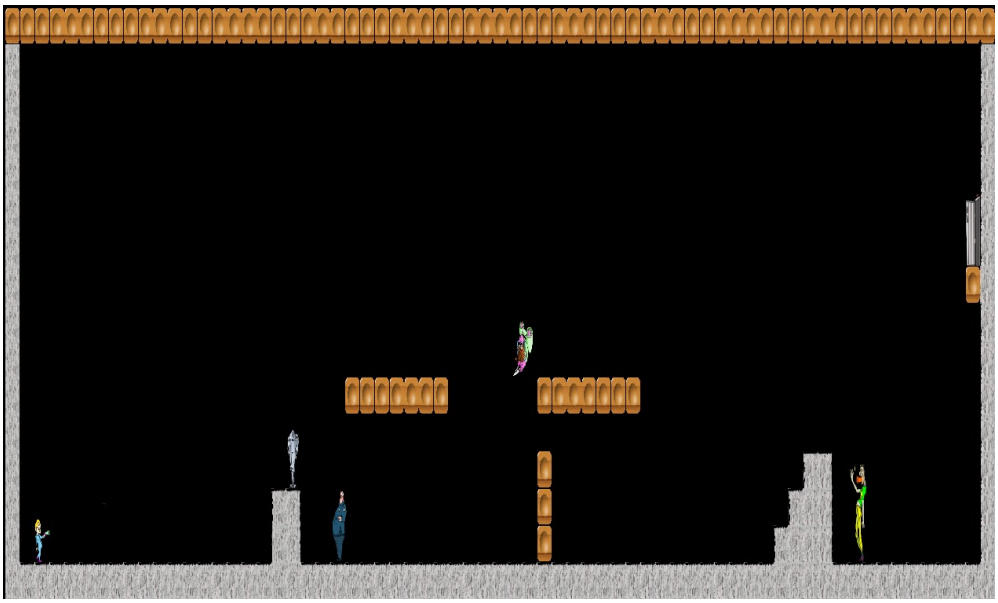
En primer lugar se delimitan las dimensiones del escenario según la cantidad de líneas del fichero y su longitud, luego, se va leyendo carácter a carácter guardándose en la matriz del Mapa los tiles y creándose los enemigos.

La estructura de los ficheros es la siguiente: en primer lugar hay líneas que comienzan por '#', esto son comentarios explicativos y simplemente se saltan, el límite de munición viene dado aquí, si la línea comienza con el símbolo '+', se toma el resto de línea como el número máximo de bloques de metal disponibles en el nivel; lo mismo pasa con el signo '-' y los bloques nube. Si el fichero contiene una línea que comienza con el símbolo '\', significa que hay un boss, y cuál es viene dado por un identificador en esa misma línea. Luego viene el mapa en sí, representado de la siguiente forma:

```
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
D                                                                       C
D                                                                       C
D                                                                       C
D                                                                       C
D                                        c                              C
D                                                                      mC
D                                                                      IA
D                                                                       C
D                                                                       C
D                      IIIIIII      IIIIIII                             C
D                                                                       C
D                   t                  I                 EF             C
D j                 EF                 I                 EGD            C
D                   CD        g        I                 EGAD      l    C
HBBBBBBBBBBBBBBBBGHBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBGAAHBBBBBBBBBBG
```

Ejemplo de mapa tal y como se guarda en la fichero. De este modo, el editor de texto sirve también como editor de mapas.

Las letras mayúsculas representan los tiles o baldosas que conforman la parte sólida del escenario, como el suelo o las paredes; los espacios en blanco son huecos por los que las criaturas pueden moverse y las letras minúsculas son los elementos.



Resultado visible del mapa anterior después de cargarse completamente.

Mapa tiene también varios métodos exclusivos para jugar en red ejecutados sólo con la llamada al método *prepararLAN()*. Por ejemplo, EstadoJugando cargará el mapa y comenzará la partida, mientra que EstadoServidor y EstadoCliente, una vez cargado el mapa y antes de comenzar, llamarán al método mencionado. Éste invoca a su vez los métodos *crearJugadores()*, que clona el jugador principal tantas veces como jugadores haya; *enumerarElementos()*, que establece un

identificador único a cada elemento con el fin de que cliente y servidor actualicen el mismo, así como también despierta a todas las criaturas por razones que se explicarán más adelante; y finalmente se invoca al método *asignarObjetivosGuardias()*, con el que se informa a las criaturas que persiguen (como los guardias, sondas y soldados) de que hay varios jugadores.

ManagerRecursos

El mánager de recursos extiende de ResourceManager y es, con diferencia, la clase más extensa de todas; no por su complejidad, sino porque se dedica a cargar todos los recursos necesarios para los escenarios, así como a cargar cada mapa en sí mismo. Funciona de la siguiente manera: en primer lugar carga todas las imágenes de todas las animaciones de cada tipo de elemento de juego y hace una instancia de cada uno, posteriormente, cuando va a cargar un mapa mediante el método *loadMap()*, va leyendo todo el fichero carácter a carácter guardando los tiles (letras mayúsculas) en una matriz y cada vez que encuentra un elemento (letras minúsculas) lo clona de las instancias originales insertando al clon en una lista mediante el método *addSprite()*. Cuando encuentra un elemento que dispara, como una torreta, clona el proyectil correspondiente y lo introduce también, asociándolos posteriormente.

TileMapRenderer

Esta clase de ejemplo tiene la funcionalidad de dibujar el escenario por pantalla. Se ha utilizado como base para hacer un renderizador propio, aprovechando así sus métodos para convertir tiles en píxels y viceversa.

RenderizadorMapas

Éste extiende de la clase anterior, y al igual que aquélla, se encarga de dibujar el fragmento de escenario que sea necesario. Tiene dos métodos: *draw()* y *pintarLAN()*. El primero, más sencillo, funciona del siguiente modo: en primer lugar se calcula el desplazamiento del jugador (esto sirve para que la imagen siempre esté centrada en él), y posteriormente se dibuja el escenario en este orden: el fondo, los tiles, el jugador, las criaturas, los bloques y finalmente la HUD, información sobre la munición. El otro método sirve para dibujar el mapa en partidas en red; es muy similar al primero, con unas ligeras variaciones: en vez de pintar un único jugador se pintan varios, obtenidos de una lista; el mapa se centra, en un principio, sobre el jugador que corresponda; sin embargo, si éste ha muerto se centra sobre el primer jugador vivo de la lista mencionada. La otra diferencia con respecto al primer método es la HUD; además de mostrar la información de la munición aparecen los nombres de los personajes encima de cada uno, un mensaje si el jugador está muerto y, si se mantiene presionada una tecla (el tabulador), también sale la puntuación de cada jugadores.

El renderizador de mapas se encarga también de despertar a las criaturas conforme aparecen por primera vez en pantalla, ahorrando de este modo tiempo computacional innecesario en actualizar y pintar. De este modo, las criaturas se van teniendo en cuanta conforme el jugador avanza en la pantalla.

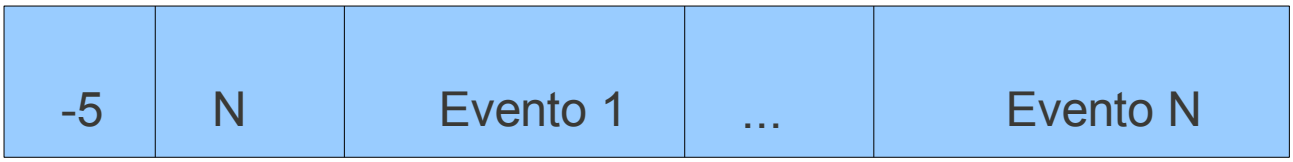
```
Iterator i = map.getSprites();
while (i.hasNext()) {
   Elemento e = (Elemento)i.next();
   int x = Math.round(e.getX()) + offsetX;
   int y = Math.round(e.getY()) + offsetY;
   e.pintar(g, offsetX, offsetY);

   // wake up the creature when it's on screen
   if (e instanceof Criatura &&
   ((x >= 0 && x < screenWidth) || (x+e.getWidth() >= 0 && x+e.getWidth() < screenWidth))
   && ((y >= 0 && y < screenHeight) || (y+e.getHeight() >= 0 && y+e.getHeight() <
   screenHeight)) &&
      !((Criatura)e).estaDespierta() && !(e instanceof EnemigoProyectilTorreta))
   {
      ((Criatura)e).wakeUp();
   }
}
```

Una vez se pinta el elemento (si es una criatura solo se pinta si está despierta) se despierta la criatura si procede; la condición es: el objeto es una criatura, el objeto está en pantalla (aunque sea parcialmente), no está despierto y no es un objeto tipo proyectil (éstos son gestionados por los objetos que los diparan, como torretas o bosses).

Un problema surge con este sistema de despertar las criaturas cuando se está en una partida en red: sólo se tiene en cuenta el avance de un jugador, pues es muy costoso comprobar si cada criatura ha aparecido en la pantalla de cada individuo; de modo que si alguien que no sea este jugador se adelanta en la pantalla se encontrará con todas las criaturas inactivas. Para solucionar esto, la clase Mapa se encarga de despertarlos a todos nada más empezar.

4.2 - Implementación del protocolo de red

Para la comunicación entre el cliente y el servidor se ha establecido un protocolo que permite el intercambio de mensajes tanto en UDP como en TCP mediante la clase Serializable EventoJuego. Esta clase tiene una estructura concreta que, al enviar por la red en UDP, se empaqueta mediante la clase PaqueteEventos en el emisor y se desempaqueta con la misma clase en el receptor.

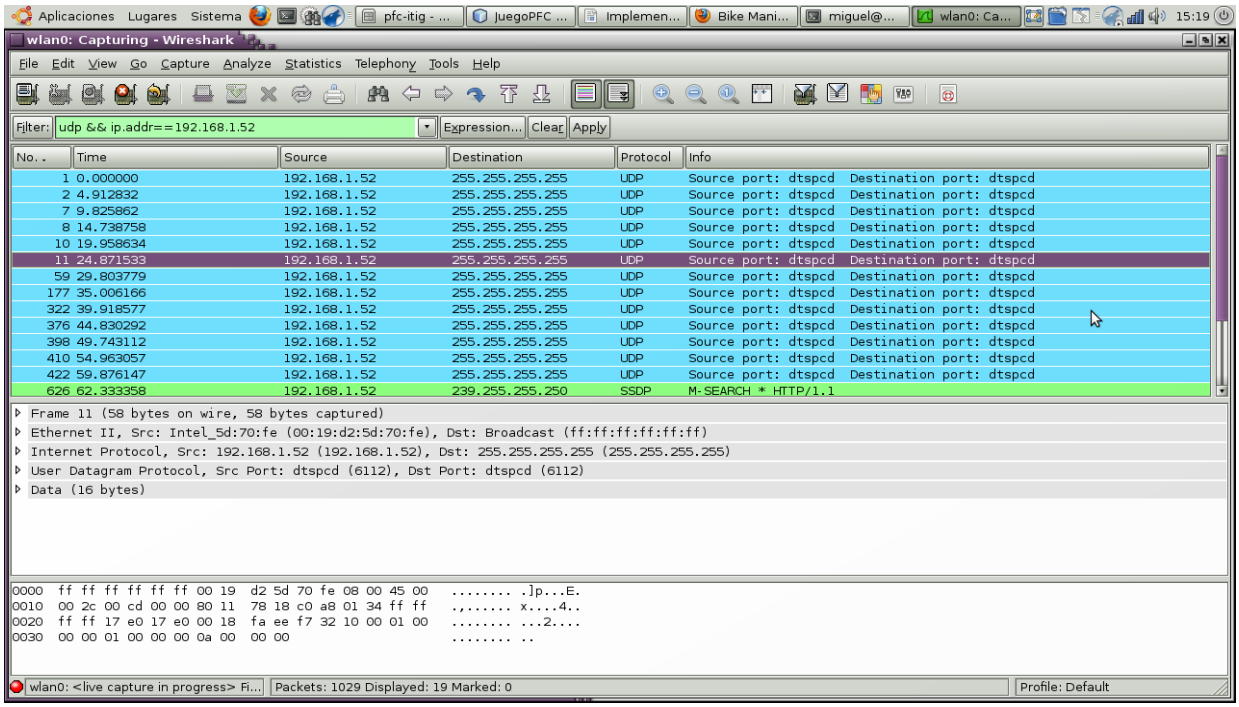


Estructura del paquete enviado por el servidor y el cliente a través de UDP.

El primer elemento del paquete sirve para identificarlo como propio del juego, debe ser el entero -5; a continuación viene un entero indicando el número de eventos que contiene el paquete, y que por tanto, han de ser leidos.

Todos los eventos tienen un tipo, indicando con ello para qué sirve la información que tienen en sus otros atributos. De este modo, dos eventos serán tratados de forma diferente si son de distinto tipo, y aunque cada tratamiento requiera leer el mismo campo, sus contenidos y efectos serán diferentes. De todo esto se encargan los métodos "tratarEvento()" de las clases EstadoServidor y EstadoCliente.

Una vez explicado el sistema de comunicación, se tratará a continuación el establecimiento de la conexión. Éste se basa en el funcionamiento del juego Warcraft III, estudiado mediante la captura de paquetes con WireShark.



El servidor del Warcraft III envía paquetes UDP a broadcast cada 5 segundos.

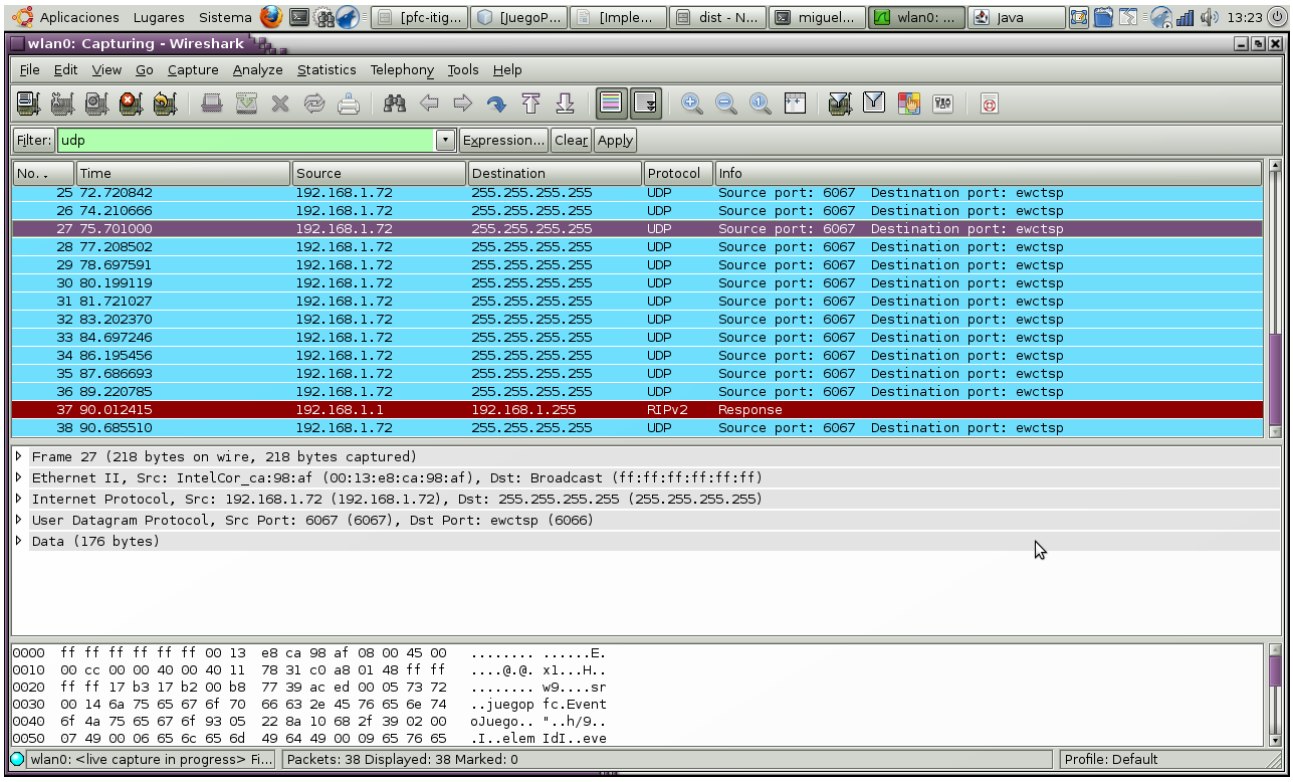
CREACIÓN DE LA PARTIDA:

En primer lugar se debe crear una partida; para ello, el jugador selecciona el tipo de partida (cooperativa o todos contra todos) y el número de clientes que se conectarán en el EstadoMenuLAN. Hecho esto se pasa al EstadoMenuCreando, donde cada cierto tiempo (1,5 segundos) se envían eventos a broadcast con el tipo S_PARTIDA_CREADA e información sobre el tipo de partida, el creador, el número de jugadores conectados y el máximo permitido. De forma paralela, se ejecuta un hilo (Thread) que se pone a escuchar peticiones de conexión de clientes, es la clase EscuchaParidasServidor.

Cuando un evento del tipo C_CONEXION llega al servidor, se leen los datos del cliente, se comprueba su nombre (si alguien se llama igual se rechaza su conexión) y se le contesta con un evento del tipo S_NOMBRE_LIBRE ó S_NOMBRE_COGIDO. Si al servidor le llega un evento

del tipo C_DESCONEXION, cierra la conexión con el jugador que lo envía, dejando un hueco libre para otro. De este modo, se tolera el cambio de conexiones dinámico antes de comenzar la partida, pero no durante la misma.

Una vez el número de jugadores conectados ha alcanzado su máximo, el servidor puede comenzar la partida enviando un evento S_COMIENZA_PARTIDA y pasando al EstadoServidor.



Wireshark capturando, al igual que con el Warcraft, paquetes UDP a broadcast cada 1,5 segundos.

BÚSQUEDA DE PARTIDAS CREADAS:

Un jugador puede buscar partidas que ya están creadas para luego conectarse a las mismas desde el EstadoMenuLAN. Para ello se ejecuta un hilo que durante cinco segundos escucha eventos del tipo S_PARTIDA_CREADA mediante la clase EscuchaPartidasCliente y guarda la información de cada partida. Una vez finalizan los cinco segundos, se crea una lista de botones con los que se selecciona la partida.

CONEXIÓN A UNA PARTIDA:

Tras obtener el listado de partidas disponibles, el jugador selecciona y se conecta a una de las partidas. Al conectarse se envía un evento del tipo C_CONEXION y se espera la respuesta del servidor, si es positiva (evento S_NOMBRE_LIBRE) se espera a que el resto de jugadores se conecten. Siempre se puede salir del EstadoMenuLAN y cerrar la conexión enviando un evento C_DESCONEXION.

Una vez conectado, la clase EscuchaPartidasCliente se vuelve a ejecutar en un hilo aparte, esta vez para atender el evento S_COMIENZA_PARTIDA; cuando éste llega, se pasa al EstadoCliente.

COMUNICACIÓN DURANTE LA PARTIDA:

Durante el transcurso del juego se deben enviar muchos eventos de forma ágil, dinámica e ininterrumpida, es por ello que casi todos los eventos se envían a través de UDP, sin embargo, hay unos eventos concretos que solo pueden enviarse una vez, pero es de vital importancia que lleguen a su destino.

EVENTOS TCP

Sólo son enviados por el servidor; el cliente los recibe en un hilo aparte, mediante la clase EscuchaTCP. Estos eventos son tres: S_MISMO_MAPA, para indicar a los clientes que deben volver a cargar el mismo mapa; S_NUEVO_MAPA, para avisar de que hay que cargar el siguiente mapa; y S_DESCONEXION, para avisar a los clientes de que la partida va a acabarse y las conexiones se van a cerrar.

EVENTOS UDP

Estos eventos son enviados tanto por el servidor como por los clientes. Cada uno se deposita al final de una cola de salida (clase ColaEventos) y al final son empaquetados en un datagrama mediante la clase PaqueteEventos y enviados por un DatagramSocket. Los datagramas son recibidos usando la clase EscuchaEventos, se desempaquetan usando de nuevo la clase PaqueteEventos y se introducen en una cola de entrada (ColaEventos). En cada iteración, tanto el EstadoServidor como el EstadoCliente, sacan de la cola un número máximo de eventos y los tratan dependiendo del tipo.

Los eventos que genera el cliente, indicados con la forma C_*, se envían al servidor para indicar las pulsaciones de las teclas; mientras que los eventos que envía el servidor, de la forma S_*, le indican al cliente los cambios en los elementos del juego: la posición y el estado de cada criatura, la munición gastada hasta el momento y la puntuación de cada jugador.

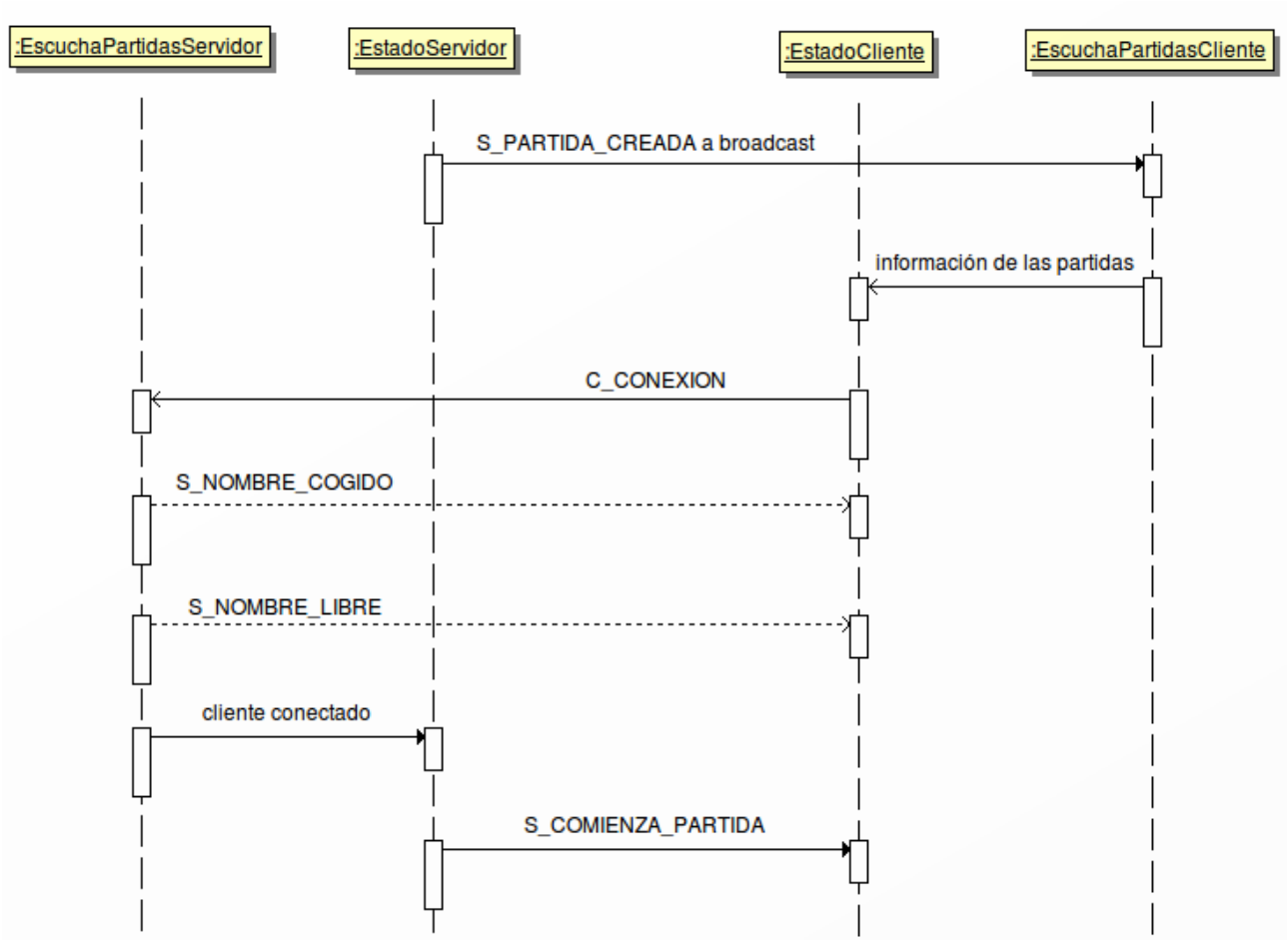


Diagrama de secuencia esquemático desde la creación de partidas y la conexión de los clientes hasta el comienzo del juego.

5 – PRUEBAS

En este apartado se muestran diferentes resultados de rendimiento de la aplicación. Este rendimiento está calculado a partir del tiempo que le cuesta al programa realizar las acciones de procesar y pintar durante el juego, tanto al jugar un único jugador como varios en red.

5.1 - Cambios de implementación

Para la realización de las pruebas se usan tres ficheros (para el tiempo de pintar, el de procesar y el total). En cada uno se almacenan cincuenta números representando cincuenta segundos de conteo; como se pinta y procesa varias veces por segundo, cada número es la media de todas las cuentas del segundo. Para poder realizar todo esto se añadirán las siguientes variables a la clase GameCore:

```
private long transcurrido=0;
protected long[] tiempoIteracionTotal=new long[50];
protected long[] tiempoIteracionUpdate=new long[50];
protected long[] tiempoIteracionPintar=new long[50];
protected long[] tablaMediasTotal=new long[200];
protected long[] tablaMediasUpdate=new long[200];
protected long[] tablaMediasPintar=new long[200];
private int mediaActual;
private int iteraciones=0;
```

donde *transcurrido* sirve para saber cuántos milisegundos han pasado (cuando llega a mil, se hacen las medias), las tablas de longitud 50 contienen la media de cada segundo, calculada a partir de las tablas correspondientes de longitud 200, que almacenan el tiempo transcurrido por iteración. La variable *mediaActual* contiene el número de medias que se han hecho en el último segundo (como mucho 200), y finalmente *iteraciones* lleva el número de segundos transcurridos (como mucho cincuenta).

Además, se debe tener cierto control sobre cuándo empezar y parar de contar, ya que sólo se quiere saber el rendimiento cuando se juega, no cuando se navega entre los menús. Por ello se debe declarar otra variable, booleana, que indique cuándo contar:

```
public static boolean cuentaTiempo=false;
```

este campo es accesible desde fuera de la clase, por ser estático. De este modo podemos asignarle el valor verdadero al empezar al jugar, en el método *start()* de la clase EstadoJugando, y volver a darle el valor falso al sair de la clase, con el método *stop()*. Igualmente lo podemos hacer con las clases

EstadoServidor y EstadoCliente.

Una vez establecidas todas las variables, se debe cambiar el código del bucle principal para poder contar:

Bucle original:

```
calcularTiempo();
procesar();
pintar();
sleep();
```

Bucle modificado:

```
calcularTiempo();
procesar();
calcularTiempoProcesamiento();
pintar();
calcularTiempoPintar();
sleep();
hacerMedias();
```

Para calcular el tiempo transcurrido entre dos puntos (por ejemplo, antes y despues de procesar) basta con restar el tiempo final menos el inicial:

```
if (cuentaTiempo && mediaActual<tablaMediasUpdate.length) {
        long aux=System.currentTimeMillis()-currTime;
        if (aux>0) {
                tablaMediasUpdate[mediaActual]=aux;
        }
}
```

en este caso, la llamada a *currentTimeMillis()* devuelve el tiempo actual, calculado después del procesamiento, mientras que la variable *currTime* contiene el tiempo antes; en aux se guarda la diferencia.

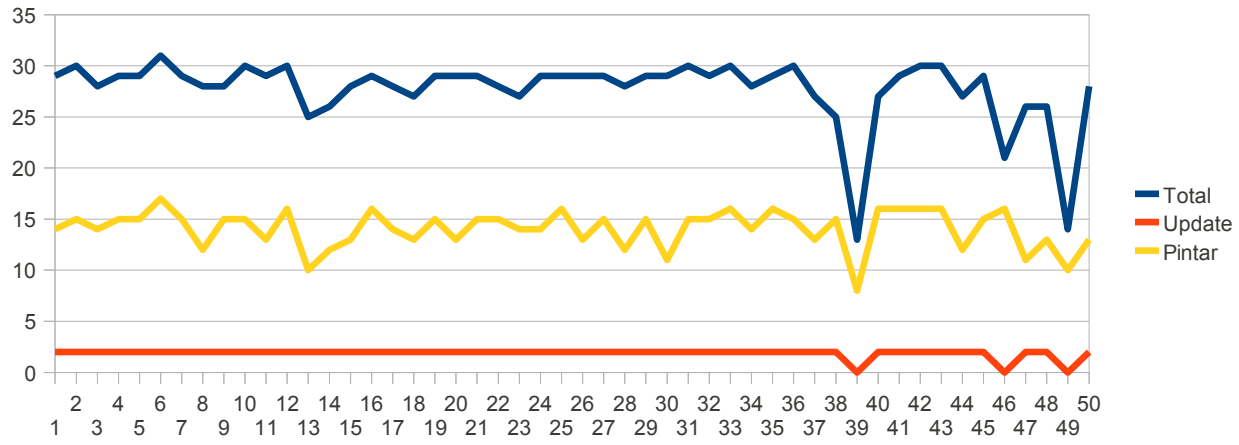
Para hacer las medias de todo el segundo hay que sumar todos los tiempos calculados y dividirlos entre el número de veces que se han calculado; siguiendo en el caso del procesamiento:

```
resultado=0;
for (int i=0;i<mediaActual;i++) {
        resultado+=tablaMediasUpdate[i];
}
resultado/=mediaActual;
if (resultado>1) {
        tiempoIteracionUpdate[iteraciones]=resultado;
}
```

Finalmente, al salir del juego se abren o crean los tres ficheros y se escribe en cada uno de ellos los cincuenta resultados correspondientes. El resultado final debe cumplir, de forma aproximada (calcular el tiempo también requiere tiempo), la siguiente fórmula:

T. TOTAL = T. PROCESAMIENTO + T. PINTAR + T. SLEEP

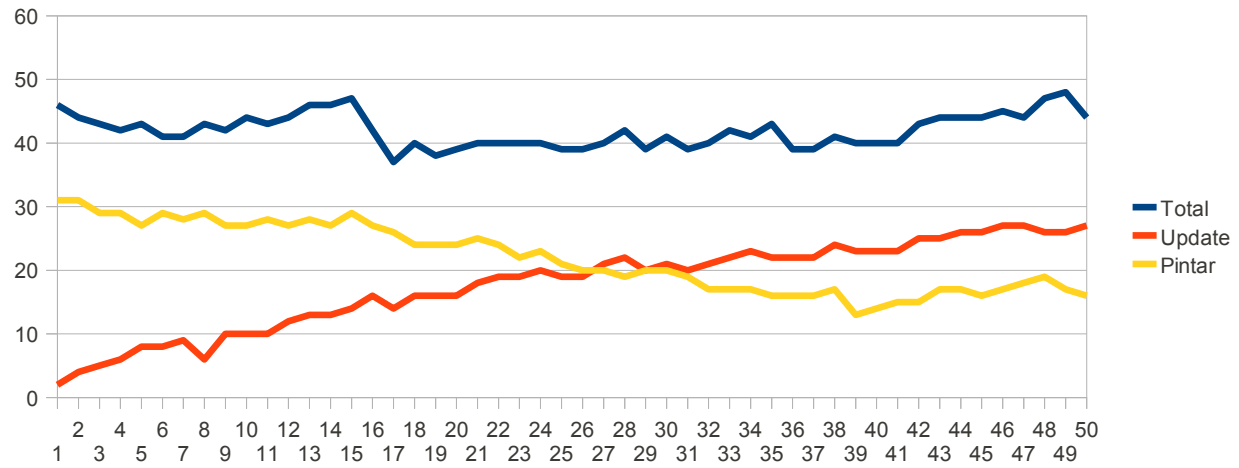
A continuación se muestra una gráfica creada a partir de los datos obtenidos del modo de juego de un jugador.



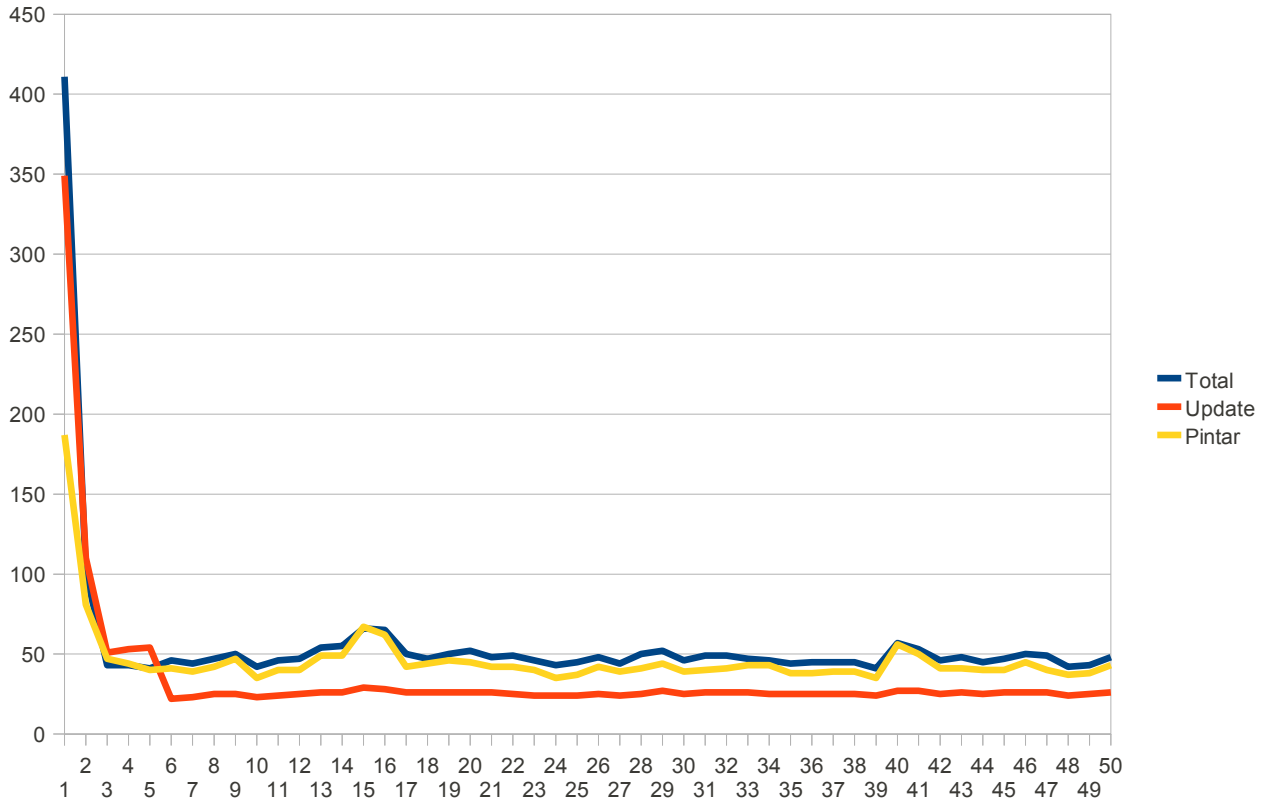
Como se puede observar, el tiempo de procesamiento (update) se mantiene prácticamente constante a lo largo del tiempo, mientras que la parte de pintar es más variable, afectando con esto al tiempo total. Esta variabilidad se debe a la máquina virtual de java y su forma de procesar los gráficos.

5.2 - Resultados de Tiempo

A continuación se muestran diferentes gráficas con resultados de juego de uno y varios jugadores en una y varias máquinas.

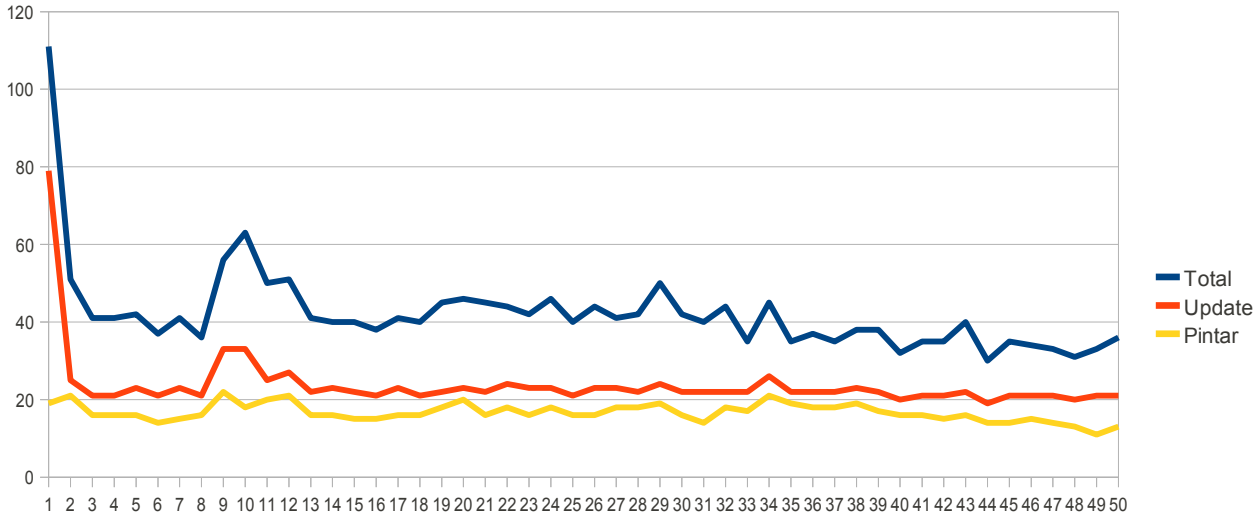


Juego de un jugador en un ordenador menos potente que el de la gráfica anterior.

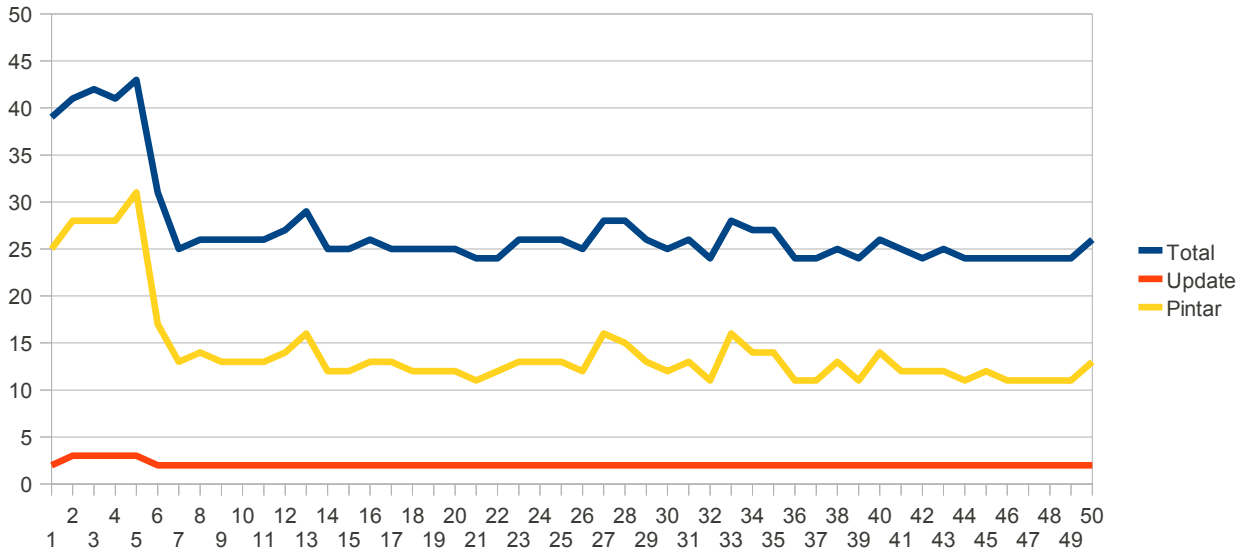


Juego de un jugador en un ordenador aún menos potente.

Como se puede observar, el rendimiento del ordenador influye profundamente en la fluidez del juego. Aunque a estos niveles no sea prácticamente perceptible y la jugabilidad siga siendo igual, si la complejidad del juego se aumentara, ya sea poniéndole gráficos en 3D, escenarios más grandes, etc., entonces sí se notaría la diferencia de un ordenador con respecto a otro.



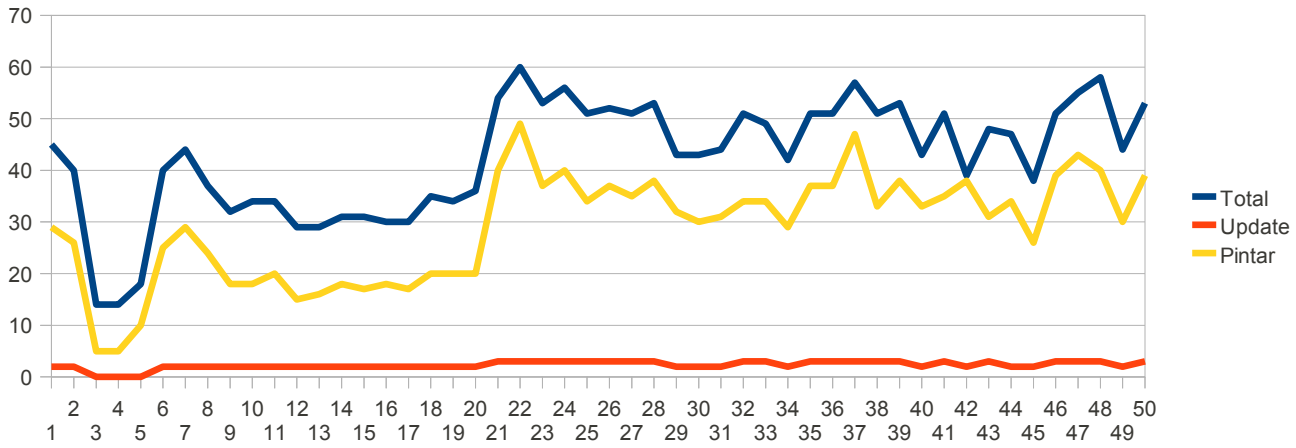
Servidor jugando en el mismo ordenador que el de la gráfica anterior.



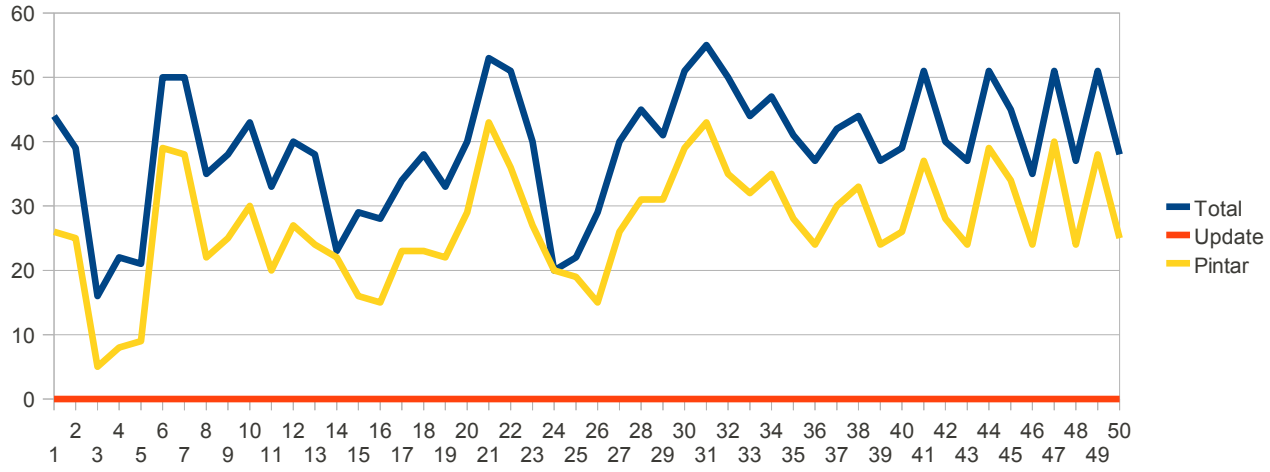
Cliente jugando la misma partida que el servidor anterior, pero en un ordenador más potente.

Comparando las gráficas se puede obtener la siguiente conclusión: el tiempo es, aproximadamente, el mismo para cada modo de juego. Según las capacidades del ordenador el rango variará, por ejemplo, en el más potente los valores, tanto para el modo un jugador como para el cliente, se mantienen en torno a los 2-3, 12-15 y 25-30 milisegundos para procesar, pintar y el tiempo total respectivamente; mientras que en el ordenador menos potente estos valores giran en torno a los 20-25, 20-40 y 40-50 milisegundos. De modo que se puede decir que la aplicación es constante en cuanto a los modos de juego se refiere.

A continuación se muestran los resultados de un servidor y un cliente jugando simultáneamente en la misma máquina:



Servidor en un ordenador potente.



Cliente simultáneo en el mismo ordenador.

Extrañamente a lo que se podía esperar, el tiempo que cuesta pintar ha dejado de ser unos 15 milisegundos para pasar a valer, de media, alrededor de 30 milisegundos, aumentando como consecuencia el tiempo total. Esto se debe a que ambas instancias del juego están compartiendo recursos, concretamente comparten la misma máquina virtual, haciéndole trabajar el doble, y costándole el doble de tiempo pintar.

En conclusión, aunque actualmente las diferencias entre ordenadores son imperceptibles, si se ampliara el videojuego, y sobre todo si se comercializara, se debería exigir un rendimiento mínimo concreto para que tuviera una buena estabilidad; y siempre evitando compartir una misma máquina virtual.

6 – CONCLUSIONES

Con este proyecto se han conseguido una serie de objetivos:

· Hacer un juego que, además de entretener a un individuo o a varios de forma simultánea, obliga al jugador a pensar y a explorar diferentes posibilidades.

· Demostrar que un lenguaje como Java es tan útil para la programación de juegos, al menos a mediana escala, como cualquier lenguaje compilado tal como C++, con la ventaja adicional de la portabilidad.

Su creación ha resultado costosa y laboriosa, pero ha sido una experiencia muy agradable, pues ver un trabajo terminado resulta siempre satisfactorio. Aunque este proyecto se podría haber extendido mucho más, por falta de tiempo y recursos se ha dejado tal como está ahora. Varias ideas que han quedado para el fururo son:

· La mejora del aspecto del juego, añadiendo más niveles, enemigos, agua, vídeos, una trama más profunda, etc.

· Profundizar el estudio de rendimiento, utilizando para ello más ordenadores, más clientes por partida, empeorando la red, etc.

· Aislar por un lado el motor de juego y por otro la parte de red con el fin de poder reutilizarlos tanto para crear otros juegos, como para emplearlos de modo didáctico (en prácticas de alguna asignatura, por ejemplo).

Si bien es cierto que el trabajo ha concluido, también hay que indicar que han quedado ciertos errores o 'bugs' que no se han podido resolver, tanto a nivel de funcionalidad (el volumen y la resolución no han sido completamente controlados) como de jugabilidad (las colisiones no se ajustan muy bien a las imágenes y los efectos al materializar los bloques sobre criaturas, principalmente Bosses, no se han determinado completamente bien).

En conclusión, hacer un videojuego es una tarea posible, pero no fácil; y con tiempo, recursos y un equipo de desarrollo de varias personas, se puede crear un producto de calidad capaz de llegar al mercado.

7 – BIBLIOGRAFÍA

- DEVELOPING GAMES IN JAVA. David Brackeen (ver siguiente enlace).
- http://www.brackeen.com/javagamebook/
- http://java.sun.com/j2se/1.5.0/docs/api/
- http://www.forodejava.com/index.php
- http://es.wikipedia.org/wiki/Historia_de_los_videojuegos
- http://es.wikipedia.org/wiki/Atari
- http://es.wikipedia.org/wiki/Tennis_for_Two
- http://www.mailxmail.com/curso-videojuegos/genero-videojuego
- http://es.wikipedia.org/wiki/G%C3%A9nero_de_videojuegos
- Algunas imágenes y datos han sido obtenidos a partir del buscador Google.